MŰEGYETEM 1782

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

A Survey on CEGAR-based Model Checking

MASTER'S THESIS

Author
Ákos HAJDU

Supervisors
Tamás TÓTH, András VÖRÖS

December 20, 2015

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Hajdu Ákos*, szigorló hallgató kijelentem, hogy ezt a diplomatervet 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, 2015. december 20.

Hajdu Ákos
hallgató

Kivonat

Napjainkban a formális verifikáció a hibamentesség és a specifikációnak való megfelelőség igazolásának egyre gyakrabban alkalmazott eszköze, különösen a biztonságkritikus rendszerek területén, ahol ezen tulajdonságok matematikailag precíz bizonyítására van szükség. A formális módszerek hátránya azonban a nagy számítási igényük. Ez az egyik legelterjedtebb formális verifikációs technikára, a modellellenőrzésre is igaz. A modellellenőrzés célja, hogy automatikusan bizonyítsa egy rendszer helyességét a lehetséges viselkedéseinek kimerítő (explicit vagy szimbolikus) vizsgálatával. Gyakran azonban kisméretű rendszerek is rendelkezhetnek kezelhetetlenül sok vagy akár végtelen számú viselkedéssel. Bár léteznek megoldások ezen probléma leküzdésére, a modellek komplexitásának növekedésével mindig újabb és hatékonyabb algoritmusokra van szükség.

Az ellenpélda-alapú absztrakció finomítás (Counterexample-Guided Abstraction Refinement, CEGAR) módszere egy gyakran alkalmazott technika az előbbi probléma leküzdésére. A CEGAR-alapú algoritmusok a rendszer modelljének egy absztrakcióján dolgoznak, amely kisebb számítási kapacitással is kezelhető. Egy elterjedt technika a lehetséges viselkedések felülbecslése a modell bizonyos kényszereinek egyszerűsítésével. Az absztrakció durvasága miatt azonban a modellellenőrzés eredménye pontatlan lehet. Ilyen esetekben az absztrakció finomítására van szükség. A CEGAR-alapú megközelítések általában egy durva absztrakcióval kezdenek, hogy minimalizálják a számítási igényüket és ezt finomítják addig, amíg el nem érik a pontos eredményt.

Diplomamunkámban megvizsgálom a tranzíciós rendszerek CEGAR-alapú modellellenőrzésének elméleti hátterét és a kapcsolódó szakirodalmat. A rendszerek és a velük kapcsolatos követelmények formalizálására bemutatom az elsőrendű- és temporális logikákat valamint megvizsgálok két konkrét CEGAR-alapú algoritmust, amely felülbecslésen alapszik. Emellett számos, a CEGAR hatékony megvalósítását lehetővé tevő technikát is bemutatok, többek között a SAT/SMT-megoldást, az interpolációt és a „lusta” absztrakciót. Ezen algoritmusok és technikák előnyeit ötvözve elkészítek egy új algoritmust. A bemutatott módszerek implementációját is elkészítettem annak érdekében, hogy kiértékelhessem és összehasonlíthassam a különböző technikák teljesítményét. A különféle modelleken elvégzett mérések elemzései rávilágítanak az algoritmusok előnyeire és hátrányaira.

Abstract

Formal verification is becoming more prevalent, especially in the development of safety-critical systems, where mathematically precise proofs are required to ensure suitability and faultlessness. However, a major drawback of formal methods is their high computational complexity. This also holds for model checking, one of the most prevalent formal verification techniques. Model checking aims to automatically verify a system by exhaustively (explicitly or symbolically) analyzing its possible behaviors. However, relatively small systems can have an unmanageably large or even infinite number of behaviors. There are several existing approaches to handle this problem, but as the complexity of the models increases, new and more efficient algorithms are required.

A widely used technique to overcome the former problem is Counterexample-Guided Abstraction Refinement (CEGAR). CEGAR-based approaches work on an abstraction of the model, which is computationally easier to handle. A common abstraction scheme is to over-approximate the set of behaviors by systematically relaxing constraints in the model. However, the result of the algorithm may be imprecise due to the coarseness of the abstraction. In such cases, the abstraction has to be refined. CEGAR-based approaches usually start with a coarse abstraction to minimize computational cost and apply refinement until a precise result is obtained.

In my work I examine the literature and the theoretical background of model checking of transition systems using CEGAR-based approaches. I present first order and temporal logic for formalizing systems and requirements, and I analyze two CEGAR algorithms based on different subtypes of over-approximation. I also examine a handful of related techniques that can make CEGAR more efficient, including SAT/SMT solving, interpolation and lazy abstraction. I also propose a new algorithm that combines the advantages of these approaches and techniques. I have implemented these algorithms in order to evaluate and compare their performance. Analysis of the measurement results highlights the advantages and shortcomings of the algorithms for several types of models.

Chapter 1

Introduction

"Society is increasingly dependent on dedicated computer and software systems to assist us in almost every aspect of daily life. Often we are not even aware that computers and software are involved. [...] Therefore a main challenge for the field of computer science is to provide formalisms, techniques, and tools that will enable the efficient design of correct and well-functioning systems despite their complexity." [1]

As Baier et al. states in [1], the trust and reliance on properly functioning hardware and software systems is rapidly increasing. Such systems can be found in safety-critical environments as well, where malfunction or failure can lead to serious damage (e.g., airplane control systems) or financial consequences (e.g., security protocols). The currently applied techniques for verifying these systems are usually testing and simulation, where the product or the prototype is given some input and its output is checked against the expected behavior. However, testing and simulation can only prove the presence of faults, but not their absence. Unless every possible input combination is checked (which is typically impossible in practice), it is not sure whether some erroneous executions are missed. These techniques are effective in the early phase of development to quickly catch faults, but their effectiveness rapidly decreases as more and more subtle errors have to be discovered. Also, as the complexity of systems grows, the number of hardly reproducible errors increases.

An attractive approach aiming to solve this problem is formal verification. Formal verification techniques have a sound mathematical basis and can prove the correctness of the system with mathematical precision. A widely studied formal verification technique is model checking, which aims to determine whether the model of a system meets a given requirement by exhaustively analyzing all possible behaviors of the system. However, a major drawback of formal methods (including model checking) is their high computational complexity. The set of possible states and behaviors of a system can become unmanageably large or even infinite. Therefore, explicitly enumerating all behaviors is just as impossible in practice as testing each input combination. Many approaches have been proposed to overcome the so-called “state space explosion” problem, including partial order reduction, symbolic methods, bounded model checking and abstraction-based techniques. Among these, my thesis focuses on abstraction.

Abstraction is a general mathematical approach for solving hard problems. In the context of model checking, abstraction means to work on a less detailed representation of the state space. My thesis focuses on existential abstraction, which over-approximates the set of possible behaviors by systematically relaxing constraints in the system. Therefore, if the abstract model meets a requirement, then it also holds for all behaviors of the original model. However, a behavior violating the requirement (i.e., a counterexample) can be caused by the coarseness of the abstraction and consequently, it cannot be reproduced in the original model. In such cases a finer abstraction is required. The main challenge of abstraction-based techniques is to find the proper level of abstraction, which is coarse enough to avoid state space explosion, but fine enough to admit the verification of the requirement. The so-called *Counterexample-Guided Abstraction Refinement (CEGAR)* approach is an automatic method for finding the proper level of abstraction. It starts with a coarse abstraction first and applies refinement (based on the non-reproducible counterexamples) until the requirement can be verified.

CEGAR is a general approach that has been applied to a wide variety of modeling formalisms and different types of abstractions. There are also several related techniques that work well along with CEGAR. In my thesis I examine two algorithms from the literature for CEGAR-based model checking of *transition systems*. The two approaches use two different abstraction types, namely *predicate abstraction* and *visible variables*. I also study a handful of related techniques, including *SAT/SMT solving*, *lazy abstraction* and *interpolation*, which aim to make CEGAR-based approaches more efficient. I propose a new algorithm that combines the advantages of the two abstraction types and the related techniques. In order to evaluate and compare the algorithms and the related methods, I implement them in a formal verification framework developed at the Fault Tolerant Systems Research Group of Budapest University of Technology and Economics. I evaluate the performance of the algorithms on several types of models, having small, large and infinite state spaces. Measurement results show that all approaches have advantages and shortcomings depending on the type of the model.

The thesis is structured as follows. Chapter 2 introduces the preliminaries of my contributions. Chapter 3 briefly summarizes related work in the field of model checking, especially focusing on abstraction-based techniques and CEGAR. Chapter 4 describes three CEGAR algorithms based on a common, general framework for existential abstraction. Chapter 5 presents the main architecture of the implementation and highlights some of the important details and features. Chapter 6 evaluates and compares the performance of the algorithms on several models. Finally, Chapter 7 concludes my work. An appendix is also provided to collect the various symbols, notations, abbreviations and definitions.

Chapter 2

Background

This chapter introduces the preliminaries of my thesis. Section 2.1 briefly describes the basic concepts of mathematical logic that are used throughout the thesis. Formal verification techniques (such as model checking) require both models and requirements with mathematically precise syntax and semantics. I introduce modeling formalisms in Section 2.2 and I present temporal logic for formalizing the requirements in Section 2.3. Finally, I describe the model checking problem itself in Section 2.4.

2.1 Mathematical logic

This section gives a brief introduction to propositional (Section 2.1.1) and first order logic (Section 2.1.2) that are often used to reason about system designs. I also describe some first order theories (Section 2.1.3) to formalize the behavior of data, e.g., numbers. Finally, Section 2.1.4 presents interpolation techniques that play an important role in my thesis. This section is based on the book of Bradley and Manna [2]. The reader is also referred to this book for more details on mathematical logic.

2.1.1 Propositional logic

This section presents propositional logic (PL), which is capable of reasoning about Boolean variables and formulas. I introduce the syntax and semantics of PL, the basic definitions and some important normal forms as well.

Syntax

Basic elements of propositional logic are the *truth symbols* $\top$ (true), $\bot$ (false) and the *propositional variables* (usually denoted by P or Q). A *formula* φ can be constructed from truth symbols, propositional variables and the application of the following *connectives* to a formula φ_1 or φ_2:

- $\neg\varphi_1$ (*negation*),
- $\varphi_1 \wedge \varphi_2$ (*conjunction*),
- $\varphi_1 \vee \varphi_2$ (*disjunction*),
- $\varphi_1 \rightarrow \varphi_2$ (*implication*),
- $\varphi_1 \leftrightarrow \varphi_2$ (*if and only if*).

An *atom* is a truth symbol $\top$, $\bot$ or a propositional variable. A *literal* is an atom or its negation. A *clause* is a disjunction of literals. A formula ψ is a *subformula* of φ if ψ occurs syntactically inside φ. The *size* of a formula φ (denoted by $|\varphi|$) is the total number of truth symbols, propositional variables and connectives in φ.

The precedence of connectives is the following, from highest to lowest: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. Moreover, $\rightarrow$ and $\leftrightarrow$ are right associative, e.g., $\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3$ equals to $\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)$.

Example 2.1. *The formula $\varphi = (P \wedge \neg Q) \vee \bot$ is constructed from a truth symbol $\bot$, two variables P, Q and three connectives $\wedge, \neg, \vee$. The subformulas of φ are $\{P, Q, \bot, \neg Q, (P \wedge \neg Q), \varphi\}$. Among these, P, Q, $\bot$ are atoms and P, Q, $\neg Q$, $\bot$ are literals. The size of φ is $|\varphi| = 6$.*

Semantics

An *interpretation* $\mathcal{I}$ assigns a truth value to every propositional variable, for example, $\mathcal{I} = \{P_1 \mapsto \bot, P_2 \mapsto \top, P_3 \mapsto \bot, \ldots\}$. Given a formula φ and an interpretation $\mathcal{I}$, $\mathcal{I} \models \varphi$ ($\mathcal{I}$ "models" φ) denotes that φ evaluates to true under $\mathcal{I}$, while $\mathcal{I} \not\models \varphi$ denotes that φ evaluates to false. The relation $\models$ is defined inductively as follows, where $\mathcal{I}$ is an interpretation, P is a propositional variable, φ_1, φ_2 are formulas and $\mathcal{I}[P]$ denotes the truth value of P under $\mathcal{I}$ [2]:

1. $\mathcal{I} \models \top$,
2. $\mathcal{I} \not\models \bot$,
3. $\mathcal{I} \models P$ iff $\mathcal{I}[P] = \top$,
4. $\mathcal{I} \not\models P$ iff $\mathcal{I}[P] = \bot$,
5. $\mathcal{I} \models \neg\varphi_1$ iff $\mathcal{I} \not\models \varphi_1$,
6. $\mathcal{I} \models \varphi_1 \wedge \varphi_2$ iff $\mathcal{I} \models \varphi_1$ and $\mathcal{I} \models \varphi_2$,
7. $\mathcal{I} \models \varphi_1 \vee \varphi_2$ iff $\mathcal{I} \models \varphi_1$ or $\mathcal{I} \models \varphi_2$,
8. $\mathcal{I} \models \varphi_1 \rightarrow \varphi_2$ iff, if $\mathcal{I} \models \varphi_1$ then $\mathcal{I} \models \varphi_2$,
9. $\mathcal{I} \models \varphi_1 \leftrightarrow \varphi_2$ iff $\mathcal{I} \models \varphi_1$ and $\mathcal{I} \models \varphi_2$, or $\mathcal{I} \not\models \varphi_1$ and $\mathcal{I} \not\models \varphi_2$.

Satisfiability and validity

A formula φ is *satisfiable* iff an interpretation $\mathcal{I}$ with $\mathcal{I} \models \varphi$ exists. A formula φ is *valid* (denoted by $\models \varphi$) iff $\mathcal{I} \models \varphi$ for every interpretation $\mathcal{I}$. Satisfiability and validity are the duals of each other: φ is valid iff $\neg\varphi$ is unsatisfiable.

Example 2.2. *The formula $\neg P \wedge (P \vee Q)$ is satisfiable by $\mathcal{I} = \{P \mapsto \bot, Q \mapsto \top\}$, but the formula $P \wedge \neg P$ is unsatisfiable. The formula $P \vee \neg P$ is valid, since for each interpretation either P or $\neg P$ will evaluate to true.*

Definition 2.1 (Boolean satisfiability problem). The *Boolean satisfiability problem (SAT)* is to decide whether a formula φ is satisfiable.

SAT was the first problem shown to be NP-complete[1] [3]. This means that no efficient algorithm is believed to exist regarding worst-case complexity. However, in practical scenarios, modern SAT solvers can handle problems with up to tens of millions of variables and clauses [4]. Most SAT solvers are based on the Davis–Putnam–Logemann–Loveland (DPLL) [5] and Conflict-Driven Clause Learning (CDCL) [6] algorithms. DPPL applies case-splitting at each variable and if a conflicting assignment is encountered, it backtracks to the previous case-split. CDCL avoids the re-exploration of conflicting assignments by learning the cause of the conflict as a clause. For more details on the SAT problem and SAT solving, the reader is referred to [7].

Equivalence and implication

The formulas φ_1 and φ_2 are *equivalent* if $\varphi_1 \leftrightarrow \varphi_2$ is valid, i.e., either $\mathcal{I} \models \varphi_1$ and $\mathcal{I} \models \varphi_2$ or $\mathcal{I} \not\models \varphi_1$ and $\mathcal{I} \not\models \varphi_2$ holds for every interpretation $\mathcal{I}$. A formula φ_1 *implies* the formula φ_2 if $\varphi_1 \rightarrow \varphi_2$ is valid, i.e., for every interpretation $\mathcal{I}$ with $\mathcal{I} \models \varphi_1$, $\mathcal{I} \models \varphi_2$ also holds. Equivalence and implication is denoted by $\varphi_1 \Leftrightarrow \varphi_2$ and $\varphi_1 \Rightarrow \varphi_2$ respectively [2].

Example 2.3. *An example on implication is $[(P \vee Q) \wedge \neg Q] \Rightarrow P$, which is also known as resolution and an example on equivalence is $\neg(P \wedge Q) \Leftrightarrow (\neg P \vee \neg Q)$, which is also known as one of De Morgan's rules.*

Normal forms

Normal forms are restrictions on the syntactic structure of a formula. The rationale behind normal forms is that they can simplify the task of the algorithms by reducing the full (and usually redundant) syntax and providing the necessary transformations.

Negation normal form (NNF) requires that only the connectives $\neg$, $\wedge$, $\vee$ are used and negations can only appear in literals. A formula φ can be transformed into an equivalent formula φ' in NNF using the following equivalences, where φ_1 and φ_2 are formulas [2]:

1. $\neg\neg\varphi_1 \Leftrightarrow \varphi_1$,
2. $\neg\top \Leftrightarrow \bot$,
3. $\neg\bot \Leftrightarrow \top$,
4. $\neg(\varphi_1 \wedge \varphi_2) \Leftrightarrow \neg\varphi_1 \vee \neg\varphi_2$,
5. $\neg(\varphi_1 \vee \varphi_2) \Leftrightarrow \neg\varphi_1 \wedge \neg\varphi_2$,

[1]The result was found by Cook (1971) and by Levin (1973) independently.

6. $\varphi_1 \rightarrow \varphi_2 \Leftrightarrow \neg\varphi_1 \vee \varphi_2$,
7. $\varphi_1 \leftrightarrow \varphi_2 \Leftrightarrow (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$.

Example 2.4. *The formula $\neg(P \wedge \neg Q)$ can be transformed by first using Rule 4 to obtain $\neg P \vee \neg\neg Q$, then using Rule 1 to obtain the NNF formula $\neg P \vee Q$.*

A formula in *conjunctive normal form (CNF)* is a conjunction of clauses, i.e., $\bigwedge_i \bigvee_j l_{i,j}$, where $l_{i,j}$ is a literal. CNF is an important normal form, since many SAT solvers work on a CNF input [8]. A formula φ can be transformed into an equivalent formula φ' in CNF by transforming φ first into NNF and then using the following distributivity rules [2]:

1. $(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \Leftrightarrow (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$,
2. $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \Leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$.

Example 2.5. *The NNF formula $(P_1 \wedge \neg P_2) \vee (\neg Q_1 \wedge Q_2)$ can be transformed by first using Rule 1 to obtain $[P_1 \vee (\neg Q_1 \wedge Q_2)] \wedge [\neg P_2 \vee (\neg Q_1 \wedge Q_2)]$, then Rule 2 twice to obtain the CNF formula $(P_1 \vee \neg Q_1) \wedge (P_1 \vee Q_2) \wedge (\neg P_2 \vee \neg Q_1) \wedge (\neg P_2 \vee Q_2)$.*

The problem with the transformation above is that the CNF formula φ' can be exponentially large compared to φ. However, to decide satisfiability, φ' is only required to be *equisatisfiable*. Two formulas φ and φ' are equisatisfiable if φ is satisfiable iff φ' is satisfiable.

Tseitin suggested a transformation into an equisatisfiable CNF formula φ' that has linear size increase regarding φ [9]. Each subformula ψ of φ (including φ itself) is associated with a new propositional variable P_ψ such that the truth value of P_ψ is the same as ψ. This mapping is done by the "representative" (rep) function in the following way:

1. $\mathsf{rep}(\top) = \top$,
2. $\mathsf{rep}(\bot) = \bot$,
3. $\mathsf{rep}(P) = P$ for a propositional variable P,
4. $\mathsf{rep}(\psi) = P_\psi$ for a formula ψ, where P_ψ is an unique variable for ψ.

The "encoding" (en) function is responsible for asserting the equivalence of ψ and P_ψ ($\psi \Leftrightarrow P_\psi$) as a CNF formula. This mapping is defined in the following way:

1. $\mathsf{en}(\top) = \top$,
2. $\mathsf{en}(\bot) = \top$,
3. $\mathsf{en}(P) = \top$,
4. $\mathsf{en}(\neg\psi) = (\neg P \vee \neg\mathsf{rep}(\psi)) \wedge (P \vee \mathsf{rep}(\psi))$, where $P = \mathsf{rep}(\neg\psi)$,
5. $\mathsf{en}(\psi_1 \wedge \psi_2) = (\neg P \vee \mathsf{rep}(\psi_1)) \wedge (\neg P \vee \mathsf{rep}(\psi_2)) \wedge (\neg\mathsf{rep}(\psi_1) \vee \neg\mathsf{rep}(\psi_2) \vee P)$, where $P = \mathsf{rep}(\psi_1 \wedge \psi_2)$,
6. $\mathsf{en}(\psi_1 \vee \psi_2) = (\neg P \vee \mathsf{rep}(\psi_1) \vee \mathsf{rep}(\psi_2)) \wedge (\neg\mathsf{rep}(\psi_1) \vee P) \wedge (\neg\mathsf{rep}(\psi_2) \vee P)$, where $P = \mathsf{rep}(\psi_1 \vee \psi_2)$,

7. $\mathsf{en}(\psi_1 \to \psi_2) = (\neg P \vee \neg\mathsf{rep}(\psi_1) \vee \mathsf{rep}(\psi_2)) \wedge (\mathsf{rep}(\psi_1) \vee P) \wedge (\neg\mathsf{rep}(\psi_2) \vee P)$, where $P = \mathsf{rep}(\psi_1 \to \psi_2)$,
8. $\mathsf{en}(\psi_1 \leftrightarrow \psi_2) = (\neg P \vee \neg\mathsf{rep}(\psi_1) \vee \mathsf{rep}(\psi_2)) \wedge (\neg P \vee \mathsf{rep}(\psi_1) \vee \neg\mathsf{rep}(\psi_2)) \wedge (P \vee \neg\mathsf{rep}(\psi_1) \vee \neg\mathsf{rep}(\psi_2)) \wedge (P \vee \mathsf{rep}(\psi_1) \vee \mathsf{rep}(\psi_2))$, where $P = \mathsf{rep}(\psi_1 \leftrightarrow \psi_2)$.

Given a formula φ with its subformulas S_φ, the formula $\varphi' = \mathsf{rep}(\varphi) \wedge \bigwedge_{\psi \in S_\varphi} \mathsf{en}(\psi)$ is in CNF, is equisatisfiable to φ and has size at most $30|\varphi| + 2$ compared to the original formula [2]. An optimization was proposed in [10], which yields a smaller CNF formula by considering the structure of the original formula.

Example 2.6 (from [2]). *Consider the formula $\varphi = (Q_1 \wedge Q_2) \vee (Q_3 \wedge Q_4)$. Its subformulas are $S_\varphi = \{Q_1, Q_2, Q_3, Q_4, Q_1 \wedge Q_2, Q_3 \wedge Q_4, \varphi\}$. The encoding is:*

- $\mathsf{en}(Q_1) = \mathsf{en}(Q_2) = \mathsf{en}(Q_3) = \mathsf{en}(Q_4) = \top$,
- $\mathsf{en}(Q_1 \wedge Q_2) = (\neg P_{(Q_1 \wedge Q_2)} \vee Q_1) \wedge (\neg P_{(Q_1 \wedge Q_2)} \vee Q_2) \wedge (\neg Q_1 \vee \neg Q_2 \vee P_{(Q_1 \wedge Q_2)})$,
- $\mathsf{en}(Q_3 \wedge Q_4) = (\neg P_{(Q_3 \wedge Q_4)} \vee Q_3) \wedge (\neg P_{(Q_3 \wedge Q_4)} \vee Q_4) \wedge (\neg Q_3 \vee \neg Q_4 \vee P_{(Q_3 \wedge Q_4)})$,
- $\mathsf{en}(\varphi) = (\neg P_{(\varphi)} \vee P_{(Q_1 \wedge Q_2)} \vee P_{(Q_3 \wedge Q_4)}) \wedge (\neg P_{(Q_1 \wedge Q_2)} \vee P_{(\varphi)}) \wedge (\neg P_{(Q_3 \wedge Q_4)} \vee P_{(\varphi)})$,

and $\varphi' = P_{(\varphi)} \wedge \bigwedge_{\psi \in S_\varphi} \mathsf{en}(\psi)$, which is equisatisfiable to φ and is in CNF.

2.1.2 First order logic

Boolean satisfiability has a tremendous importance in computer science. However, some problems can be expressed more naturally in richer languages. This section presents first order logic (FOL), which extends propositional logic with predicates, functions and quantifiers. Consequently, formulas in FOL can not only evaluate to truth values, but to any abstract concept, e.g., integers, animals, names.

Syntax

The basic elements of FOL are *terms* that can be *variables* $(x, y, z, \ldots)$ and *constants* $(a, b, c, \ldots)$. Complex terms are created by *functions* $(f, g, h, \ldots)$. A constant can also be regarded as a 0-ary function. Propositional variables are generalized to *predicates* $(p, q, r, \ldots)$. An n-ary predicate has n terms as arguments. A propositional variable in FOL can also be regarded as a 0-ary predicate.

Atoms, literals and formulas are generalized for FOL as well. An *atom* is $\top$, $\bot$, or an n-ary predicate applied to n terms. A *literal* is an atom or its negation. A *formula* is a literal or the application of *connectives* ($\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$) and *quantifiers*. The two quantifiers are the *existential quantifier* (denoted by $\exists x.\varphi[x]$) and the *universal quantifier* (denoted by $\forall x.\varphi[x]$). The variable x is the *quantified variable* and $\varphi[x]$ is the *scope*. The variable x is also said to be *bound* (by the quantifier). A variable is *free* in a formula φ if it has a non-bound occurrence. A formula is *closed* if it contains no free variables. If a formula φ is not closed and $x_1, x_2, \ldots, x_n$ are the free variables of φ, then the *universal closure* of φ is $\forall x_1.\forall x_2.\ldots\forall x_n.\varphi$ and the *existential closure* is $\exists x_1.\exists x_2.\ldots\exists x_n.\varphi$.

The precedence of connectives and quantifiers is the following, from highest to lowest: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\forall$, $\exists$.

Semantics

Formulas of FOL do not only evaluate to true or false, but to any value from a specified domain. An *interpretation* $\mathcal{I}$ is therefore, a pair of a *domain* $D_\mathcal{I}$ and an *assignment* $\alpha_\mathcal{I}$. The domain $D_\mathcal{I}$ is a nonempty set of abstract objects (e.g., integers, animals, names). The assignment $\alpha_\mathcal{I}$ maps

- constants to elements in $D_\mathcal{I}$,
- variable symbols x to values $x_\mathcal{I} \in D_\mathcal{I}$,
- n-ary function symbols f to n-ary functions $f_\mathcal{I} \colon D_\mathcal{I}^n \mapsto D_\mathcal{I}$,
- n-ary predicate symbols p to n-ary predicates $p_\mathcal{I} \colon D_\mathcal{I}^n \mapsto \{\top, \bot\}$.

> **Example 2.7 (from [2]).** *The formula $(x + y > z) \rightarrow (y > z - x)$ contains the binary function symbols "$+$", "$-$", the binary predicate symbol "$>$" and variables x, y, z. Since "$+$", "$-$" and "$>$" are just symbols, the formula $p(f(x, y), z) \rightarrow p(y, g(z, x))$ has the same meaning. To construct an interpretation, let $D_\mathcal{I} = \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and let "$+$", "$-$" and "$>$" be the standard addition, subtraction and greater-than relation over integers. Finally, let x, y and z be 13, 42 and 1 respectively. Thus, the assignment is $\alpha_\mathcal{I} = \{+ \mapsto +_\mathbb{Z}, - \mapsto -_\mathbb{Z}, > \mapsto >_\mathbb{Z}, x \mapsto 13, y \mapsto 42, z \mapsto 1, \ldots\}$.*

Arbitrary terms can be evaluated recursively, i.e., $\alpha_\mathcal{I}[f(t_1, t_2, \ldots, t_n)] = \alpha_\mathcal{I}[f](\alpha_\mathcal{I}[t_1], \alpha_\mathcal{I}[t_2], \ldots, \alpha_\mathcal{I}[t_n])$ for a function f and $\alpha_\mathcal{I}[p(t_1, t_2, \ldots, t_n)] = \alpha_\mathcal{I}[p](\alpha_\mathcal{I}[t_1], \alpha_\mathcal{I}[t_2], \ldots, \alpha_\mathcal{I}[t_n])$ for a predicate p. Given a FOL formula φ and an interpretation $\mathcal{I}(D_\mathcal{I}, \alpha_\mathcal{I})$ the relation $\models$ is defined inductively as follows [2], where p is a predicate and $t_1, t_2, \ldots, t_n$ are terms:

1. $\mathcal{I} \models \top$,
2. $\mathcal{I} \not\models \bot$,
3. $\mathcal{I} \models p(t_1, t_2, \ldots, t_n)$ iff $\alpha_\mathcal{I}[p(t_1, t_2, \ldots, t_n)] = \top$,
4. rules for the connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ work as in PL.

For quantifiers, let $\mathcal{I}' = \mathcal{I} \lhd \{x \mapsto v\}$ denote the x-variant of the interpretation $\mathcal{I}$, where $D_{\mathcal{I}'} = D_\mathcal{I}$ and $\alpha_{\mathcal{I}'}[y] = \alpha_\mathcal{I}[y]$ for all constant, free variable, function and predicate symbols, except for x, where $\alpha_{\mathcal{I}'}[x] = v$. Then $\models$ is defined as follows [2]:

5. $\mathcal{I} \models \forall x.\varphi$ iff for all $v \in D_\mathcal{I}$, $\mathcal{I} \lhd \{x \mapsto v\} \models \varphi$,
6. $\mathcal{I} \models \exists x.\varphi$ iff a $v \in D_\mathcal{I}$ exists with $\mathcal{I} \lhd \{x \mapsto v\} \models \varphi$.

Satisfiability and validity

Satisfiability and validity is defined similarly to PL: a formula φ is *satisfiable* iff an interpretation $\mathcal{I}$ with $\mathcal{I} \models \varphi$ exists and it is *valid* iff $\mathcal{I} \models \varphi$ holds for every interpretation $\mathcal{I}$. Technically, satisfiability and validity can only be applied to closed FOL formulas. However, by convention a non-closed formula is said to be satisfiable if its existential closure is satisfiable and it is said to be valid if its universal closure is valid. Church [11] and Turing [12] proved that satisfiability (and consequently, also validity) is undecidable for FOL in the general case.

2.1.3 First order theories

First order theories formalize the structures (e.g., numbers, lists, ...) of a domain to enable reasoning about them formally. While satisfiability in FOL is undecidable in general, it is decidable in many practical first order theories (or in their fragments). In this section I describe some theories that I use in my work. However, the CEGAR algorithms presented in Chapter 4 can handle other theories as long as the underlying solver supports them.

Definition 2.2 (First order theory). A first order theory $\mathcal{T} = (\Sigma, \mathcal{A})$ [2] is defined by

- a *signature* Σ, which is the set of constant, function and predicate symbols,
- a set of *axioms* $\mathcal{A}$, which is a set of closed FOL formulas, in which only constant, function and predicate symbols of Σ appear.

A Σ-*formula* is constructed from constant, function and predicate symbols of Σ along with variables, connectives and quantifiers. The axioms $\mathcal{A}$ provide meaning for the formulas. A Σ-formula φ is valid in $\mathcal{T}$ (or $\mathcal{T}$-valid), if every interpretation $\mathcal{I}$ satisfying the axioms also satisfies φ (i.e., if $\mathcal{I} \models \varphi_A$ for all $\varphi_A \in \mathcal{A}$, then $\mathcal{I} \models \varphi$). This is denoted by $\mathcal{T} \models \varphi$. A Σ-formula φ is satisfiable in $\mathcal{T}$ (or $\mathcal{T}$-satisfiable) if an interpretation $\mathcal{I}$ exists that satisfies the axioms and φ. A theory $\mathcal{T}$ is *complete* if for every closed Σ-formula φ either $\mathcal{T} \models \varphi$ or $\mathcal{T} \models \neg\varphi$ holds. A theory $\mathcal{T}$ is *decidable*, if $\mathcal{T} \models \varphi$ is decidable by an algorithm for every Σ-formula φ.

Equality

One of the simplest first order theories is the theory of equality ($\mathcal{T}_E$). Its signature $\Sigma_E = \{\doteq, a, b, c, \dots, f, g, h, \dots, p, q, r, \dots\}$ consists of the binary predicate "$\doteq$" (equality) and a countable number of constant, function and predicate symbols. To distinguish between equality in a theory $\mathcal{T}$ and equality in the meta-theory, I denote the former one with "$\doteq$" and the latter one with "$=$". The axioms $\mathcal{A}_E$ listed below give meaning to the equality symbol [2]:

1. $\forall x.\ x \doteq x$ (reflexivity),
2. $\forall x, y.\ x \doteq y \rightarrow y \doteq x$ (symmetry),
3. $\forall x, y, z.\ x \doteq y \land y \doteq z \rightarrow x \doteq z$ (transitivity),
4. $\forall x_1, x_2, \ldots, x_n, y_1, y_2, \ldots y_n.\ (\bigwedge_i x_i \doteq y_i) \rightarrow f(x_1, x_2, \ldots, x_n) \doteq f(y_1, y_2, \ldots, y_n)$ for each $n \in \mathbb{Z}^+$ and n-ary function symbol f,
5. $\forall x_1, x_2, \ldots, x_n, y_1, y_2 \ldots y_n.\ (\bigwedge_i x_i \doteq y_i) \rightarrow p(x_1, x_2, \ldots, x_n) \leftrightarrow p(y_1, y_2, \ldots, y_n)$ for each $n \in \mathbb{Z}^+$ and n-ary predicate symbol p.

The first three axioms state that equality is an equivalence relation, while the last two state that it is a congruence relation. I also use $x \not\doteq y$ to denote $\neg(x \doteq y)$. The theory of equality is undecidable in general, but its quantifier-free fragment is decidable [2].

Presburger arithmetic

The theory of Presburger arithmetic ($\mathcal{T}_{\mathbb{N}}$) has the signature $\Sigma_{\mathbb{N}} = \{0, 1, +, \doteq\}$, where 0 and 1 are constants, "$+$" (addition) is a binary function and "$\doteq$" (equality) is a binary predicate. The axioms $\mathcal{A}_{\mathbb{N}}$ are the following [2]:

1. $\forall x.\ \neg(x + 1 \doteq 0)$ (zero),
2. $\forall x, y.\ x + 1 \doteq y + 1 \rightarrow x \doteq y$ (successor),
3. $\varphi[0] \land (\forall x.\ \varphi[x] \rightarrow \varphi[x + 1]) \rightarrow \forall x.\ \varphi[x]$ (induction),
4. $\forall x.\ x + 0 \doteq x$ (plus zero),
5. $\forall x, y.\ x + (y + 1) \doteq (x + y) + 1$ (plus successor).

Presburger showed that $\mathcal{T}_{\mathbb{N}}$ is decidable [13]. The *theory of integers* ($\mathcal{T}_{\mathbb{Z}}$) has the signature $\Sigma_{\mathbb{Z}} = \{\ldots, -2, -1, 0, 1, 2, \ldots, -3\cdot, -2\cdot, 2\cdot, 3\cdot, \ldots, +, -, \doteq, >\}$, where $\ldots, -2, -1, 0, 1, 2, \ldots$ are constants, $\ldots, -3\cdot, -2\cdot, 2\cdot, 3\cdot, \ldots$ are unary functions representing constant coefficients, "$-$" and "$+$" are binary functions, "$\doteq$" and "$>$" are binary predicates having the obvious corresponding meaning in the domain of integers. Each formula of $\Sigma_{\mathbb{Z}}$ can be encoded in $\Sigma_{\mathbb{N}}$, hence the theory of integers is just a syntactic extension of Presburger arithmetic. *Peano arithmetic* ($\mathcal{T}_{\text{PA}}$) however, introduces multiplication, which renders it incomplete and undecidable [14].

Theory of rationals

The theory of rationals ($\mathcal{T}_{\mathbb{Q}}$) has the signature $\Sigma_{\mathbb{Q}} = \{0, 1, +, -, \doteq, \geq\}$, where 0 and 1 are constants, "$+$" (addition) is a binary function, "$-$" (negation) is an unary function, "$\doteq$" (equality) and "$\geq$" (weak inequality) are binary predicates. The axioms $\mathcal{A}_{\mathbb{Q}}$ are the following [2]:

1. $\forall x, y.\ x \geq y \land y \geq x \rightarrow x \doteq y$ (antisymmetry),

2. $\forall x, y, z.\ x \geq y \land y \geq z \rightarrow x \geq z$ (transitivity),
3. $\forall x, y.\ x \geq y \lor y \geq x$ (totality),
4. $\forall x, y, z.\ (x + y) + z \doteq x + (y + z)$ (associativity),
5. $\forall x.\ x + 0 \doteq x$ (identity),
6. $\forall x.\ x + (-x) \doteq 0$ (inverse),
7. $\forall x, y.\ x + y \doteq y + x$ (commutativity),
8. $\forall x, y, z.\ x \geq y \rightarrow x + z \geq y + z$ (ordered),
9. $\forall x.\ nx \doteq 0 \rightarrow x \doteq 0$, for each $n \in \mathbb{Z}^+$, where nx is x added to itself x times (torsion free),
10. $\forall x . \exists y.\ x \doteq ny$, for each $n \in \mathbb{Z}^+$ (divisible).

The theory of rationals is decidable by eliminating quantifiers [2] and then using linear programming [15].

Satisfiability modulo theories

Similarly to the SAT problem in propositional logic, satisfiability is also defined for first order logic with a background theory [16].

Definition 2.3 (Satisfiability modulo theories). The *satisfiability modulo theories (SMT)* problem is to decide if a Σ-formula is satisfiable in a theory $\mathcal{T} = (\Sigma, \mathcal{A})$.

The first SMT solvers transformed the SMT formula into an equisatisfiable PL formula and used an “off-the-shelf” SAT solver. This concept is called the *eager approach* [17] and it is similar to compiling a high-level program into machine code. The drawback of this method is the possibly large size of the transformed formula. Modern SMT solvers therefore, usually implement the *lazy approach* [18]. The lazy approach uses a SAT solver as the propositional core by replacing FOL atoms with propositional variables. If the propositional formula is unsatisfiable, then so is the SMT formula. On the other hand, if the SAT solver finds an interpretation, it is checked with a *theory solver.* Theory solvers are specialized in checking the consistency of a set of literals. If the interpretation is contradicting in the theory $\mathcal{T}$, then its negation is valid. Therefore, it is added to the propositional formula and the SAT solver is called again.

Example 2.8 (from [19]). *Consider the formula* $\varphi = \neg(a \geq 3) \land (a \geq 3 \lor a \geq 5)$ *with the background theory* $\mathcal{T}_{\mathbb{Q}}$*. At first, the atomic formulas are replaced by variables to obtain the propositional formula* $\varphi' = \neg P \land (P \lor Q)$*. The SAT solver yields the interpretation* $\{P \mapsto \bot, Q \mapsto \top\}$*, representing* $\neg(a \geq 3) \land a \geq 5$*. However* $\neg(a \geq 3)$ *and* $a \geq 5$ *cannot be true at the same time in* $\mathcal{T}_{\mathbb{Q}}$*. Therefore,* $\psi = a \geq 3 \lor \neg(a \geq 5)$ *must be valid, which is translated into* $\psi' = P \lor \neg Q$*. The SAT solver is now given the formula* $\varphi' \land \psi' = \neg P \land (P \lor Q) \land (P \lor \neg Q)$*, which is proved to be unsatisfiable, implying that* φ *is also unsatisfiable.*

In practical cases, an SMT problem is a combination of several theories (e.g., $1 < x \wedge x < 4 \wedge f(x) \not\doteq f(2)$). Nelson and Oppen proposed a method for deciding the SMT problem for the union of disjoint, stably-infinite, quantifier-free decidable theories [20].

2.1.4 Interpolation

This section presents Craig interpolants and interpolation sequences, which play an important role in SAT/SMT-based model checking [8] and also in my thesis (Section 4.4).

Definition 2.4 (Craig interpolant). Let α and β be FOL formulas such that $\alpha \wedge \beta$ is unsatisfiable. A formula $\mathscr{I}$ is a *Craig interpolant* (or simply an *interpolant*) for (α, β) if the following properties hold [21]:

- $\alpha \Rightarrow \mathscr{I}$,
- $\mathscr{I} \wedge \beta$ is unsatisfiable,
- $\mathscr{I}$ refers only to common symbols of α and β (excluding the symbols of the logic).

Informally this means that $\mathscr{I}$ generalizes α, but it can still contradict β. William Craig showed that an interpolant always exists for FOL formulas α and β with at least one symbol in common and $\alpha \wedge \beta$ being unsatisfiable [22]. Craig interpolants can be generated efficiently from refutation proofs both in PL [23] and several quantifier-free theories [24].

Example 2.9. *Let α be $(x \doteq y \wedge x \doteq z)$ and let β be $(y \doteq t \wedge t \not\doteq z)$. The conjunction $\alpha \wedge \beta$ is clearly unsatisfiable. The formula $(y \doteq z)$ is an interpolant for (α, β), since*

- $(x \doteq y \wedge x \doteq z) \Rightarrow (y \doteq z)$,
- $(y \doteq z) \wedge (y \doteq t \wedge t \not\doteq z)$ *is unsatisfiable,*
- $(y \doteq z)$ *contains only the symbols y and z which are present both in α and β.*

Interpolation can be extended from two formulas to a sequence of formulas in the following way.

Definition 2.5 (Interpolation sequence). Let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be a sequence of FOL formulas such that $\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n$ is unsatisfiable. A sequence of formulas $\mathscr{I}_0, \mathscr{I}_1, \ldots, \mathscr{I}_n$ is an *interpolation sequence* for $\alpha_1, \alpha_2, \ldots, \alpha_n$ if the following properties hold [25]:

- $\mathscr{I}_0 = \top$, $\mathscr{I}_n = \bot$,
- $\mathscr{I}_j \wedge \alpha_{j+1} \Rightarrow \mathscr{I}_{j+1}$ for $0 \leq j < n$,
- $\mathscr{I}_j$ refers only to common symbols of $\alpha_1, \ldots, \alpha_j$ and $\alpha_{j+1}, \ldots, \alpha_n$ for $0 < j < n$.

Example 2.10. *Let $\alpha_1 = (x > 3 \wedge x \doteq y)$, $\alpha_2 = (z > y)$ and $\alpha_3 = (z < 2)$. The conjunction $\alpha_1 \wedge \alpha_2 \wedge \alpha_3$ is clearly unsatisfiable. The formulas $\mathscr{I}_0 = \top$, $\mathscr{I}_1 = (y > 3)$, $\mathscr{I}_2 = (z > 3)$, $\mathscr{I}_3 = \bot$ form an interpolation sequence, since:*

- $\top \wedge (x > 3 \wedge x \doteq y) \Rightarrow (y > 3)$,
- $(y > 3) \wedge (z > y) \Rightarrow (z > 3)$,

- $(z > 3) \wedge (z < 2) \Rightarrow \bot$,
- *$\mathscr{I}_1$ refers to y, which is present in α_1 and α_2, α_3,*
- *$\mathscr{I}_2$ refers to z, which is present in α_1, α_2 and α_3.*

Interpolation sequences can be calculated iteratively by computing Craig interpolants for $\alpha = \mathscr{I}_{j-1} \wedge \alpha_j$ and $\beta = \bigwedge_{i=j+1}^{n} \alpha_i$ for each $1 \leq j \leq n$ using the same refutation [25].

2.2 Modeling formalisms

Formal verification techniques require models with mathematically precise syntax and semantics. There are several existing low- and high-level modeling formalisms. Most of the low-level models explicitly represent the possible states and behaviors of the system using graphs with labeled nodes or edges. Such models can be handled easily by formal verification algorithms. However, even relatively simple systems can have a complex low-level representation, which renders these formalisms unsuitable for practical use. Kripke structures and transition systems (presented in Section 2.2.1) are widely used in model checking as a low-level formalism.

High-level formalisms also have precise syntax and semantics but they offer a compact representation and constructs that are closer to the modeled domain, which makes them easier to understand and use in practice. High-level formalisms can be either based on a graphical or a textual language or they may even have both representations. Graphical formalisms (e.g., state charts, Petri nets, data flow networks) are usually based on graphs, while textual formalisms are similar to programming languages. In my thesis I use symbolic transition systems (described in Section 2.2.2). Verification algorithms usually translate high-level models to low-level before addressing the problem (see “state space generation” in Section 2.4).

2.2.1 Kripke structures

A *Kripke structure* [26] is a directed graph whose nodes and edges correspond to the states and transitions of the modeled system respectively. Furthermore, each state is labeled with a set of propositions that hold on that state.

Definition 2.6 (Kripke structure). A Kripke structure over a set of *atomic propositions* A is a tuple $M = (S, R, L, I)$, where

- S is the set of *states*,
- $R \subseteq S \times S$ is the set of *transitions*,
- $L\colon S \mapsto 2^A$ is the *labeling function* assigning each state a subset of A,
- $I \subseteq S$ is the nonempty set of *initial states*.

The structure (S, R, I) obtained by omitting the labeling is often referred to as a *transition system*. In my thesis I represent systems using FOL formulas (see Section 2.2.2), hence atomic propositions are FOL formulas over the variables of the system, e.g., $x > y$.

A *path* π is a sequence of states $\pi = (s_1, s_2, \dots)$ with $(s_i, s_{i+1}) \in R$ for each $i \geq 1$. Paths can be either finite or infinite. A *computation tree* for a state $s_0 \in S$ is a tree with states as nodes and transitions as edges. The root node is s_0 and $s \to s'$ is an edge in the tree if $(s, s') \in R$. Informally, the computation tree represents all the possible paths starting from s_0.

Example 2.11. *Figure 2.1 presents a simple Kripke structure modeling a traffic light. The model has five states* $(s_1, s_2, \dots, s_5)$*, each labeled with a subset of* $A =$ *{red, yellow, green, off}. The initial state* s_1 *is denoted by a double circle. The normal operation of the light corresponds to the infinite path* $\pi_{normal} = (s_1, s_2, s_3, s_4, \dots)$*. A behavior where the light turns off is for example the path* $\pi_{off} = (s_1, s_2, s_5, s_1)$*.*

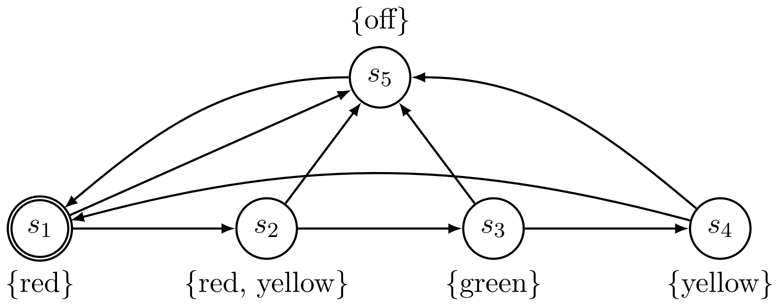


Figure 2.1: *Example Kripke structure of a traffic light.*

2.2.2 Symbolic transition systems

In my work I represent the models using symbolic transition systems, which offer a compact way of representing the set of states, transitions and initial states using variables and FOL formulas.

Definition 2.7 (Symbolic transition system). A *Symbolic transition system* is a tuple $T = (V, \mathit{Inv}, \mathit{Tran}, \mathit{Init})$, where

- $V = \{v_1, v_2, \dots, v_n\}$ is a finite set of variables with domains $D_{v_1}, D_{v_2}, \dots, D_{v_n}$,
- *Inv* (*invariant*) is a FOL formula over V, representing the *states* along with V,
- *Tran* is a FOL formula over V, representing the *transition relation*,
- *Init* is a FOL formula over V, representing the *initial states.*

A state $s \in D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$ is an assignment of the variables. Thus, s can also be denoted by enumerating the values of the variables, i.e., $s = d = (d_1, d_2, \dots, d_n)$, where $d_i \in D_{v_i}$. The models only contain variables and interpreted symbols. Therefore, given a FOL formula φ over the variables V let $s \models \varphi$ denote, that assigning the variables in φ with the values of s evaluates to true. Analogously, let $s \not\models \varphi$ denote that φ evaluates to false.

A transition system (S, R, I) can be obtained from a symbolic transition system $T = (V, \mathit{Inv}, \mathit{Tran}, \mathit{Init})$ in the following way.

- The set of states S is defined by the domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$ and *Inv* in the following way: $S = \{s \in D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n} \mid s \models \mathit{Inv}\}$. Informally, S contains all possible assignments that satisfy the invariant.
- The set of transitions R is defined by *Tran*. In the transition formula, variables have a non-primed $(v_1, v_2, \ldots, v_n)$ and a primed $(v_1', v_2', \ldots, v_n')$ version, corresponding to the actual and successor states respectively. R is then defined in the following way: $R = \{(s, s') \in S \times S \mid (s, s') \models \mathit{Tran}\}$. Informally, s' is a successor of s if assigning values from s to the non-primed variables and values from s' to the primed variables evaluates to true.
- Finally, the set of initial states I is defined by *Init* in the following way: $I = \{s \in S \mid s \models \mathit{Init}\}$. Informally, I is the subset of S for which the initial formula holds.

Example 2.12. *The TTMC framework (Chapter 5), in which I work provides a textual language for describing symbolic transition systems. A simple example can be seen in Listing 2.1. The system has an integer variable x and a Boolean r, i.e., $D_x = \mathbb{Z}, D_r = \mathbb{B}$. Inv is the conjunction of formulas labeled with the keyword* `invariant`*. In this example an invariant is used to restrict x to values in $\{1, 2, 3, 4\}$, but in general, an invariant can correspond to an arbitrary formula. Init is the conjunction of formulas labeled with* `initial`*. In this example Init assigns a single initial value to each variable, but again, it can also correspond to an arbitrary formula. Tran is the conjunction of formulas labeled with* `transition`*. It can be seen that Tran defines the relationship between the values of variables in the actual state (x, r) and the successor state (x', r'). A formula of the form* `if` φ_1 `then` φ_2 `else` φ_3 *is syntactic shortcut for $(\varphi_1 \rightarrow \varphi_2) \wedge (\neg\varphi_1 \rightarrow \varphi_3)$. Finally, a temporal logic expression (see Section 2.3) is given, which the system must satisfy. The corresponding transition system can be seen in Figure 2.2, where states are annotated with the (x, r) values.*

Listing 2.1: *Example symbolic transition system described in the TTMC framework.*

```
specification System {
    property safe : {
        local var x : integer
        local var r : boolean

        invariant x >= 1 and x <= 4

        initial x = 1
        initial r = false

        transition x' = (
            if x < 4 and not r then x + 1
            else 1
        )

        transition (r' = true and x = 2) or r' = false
    } models G(not r or not x = 2)
}
```

Since the invariant, the transition and the initial conditions are all FOL formulas, an SMT solver can be used to evaluate standard queries about the model. Some examples are listed below.

- Given a state $s \in D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$, $s \in S$ can be decided by querying $s \models \mathit{Inv}$ from the solver.
- States $s \in S$ can be enumerated by querying for satisfying assignments of *Inv*.
- Given states $s, s' \in S$, $(s, s') \in R$ can be decided by querying $(s, s') \models \mathit{Tran}$ from the solver.
- Successors s' of a state $s \in S$ can be enumerated by assigning values of s to the non-primed variables of *Tran*, asserting *Inv* for the primed variables and querying for satisfying assignments.
- Given a state $s \in D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$, $s \in I$ can be decided by querying $s \models \mathit{Init} \wedge \mathit{Inv}$ from the solver.
- Furthermore, combining these queries is also possible, e.g., the possible solutions of $s_1 \in I \wedge s_2 \in S \wedge s_3 \in S \wedge (s_1, s_2) \in R \wedge (s_2, s_3) \in R$ yield paths from initial states with length three.

Example 2.13. *Consider a system with a single variable x and suppose that $\mathit{Inv} = (x < 5)$, $\mathit{Init} = (x \doteq 0)$ and $\mathit{Tran} = (x' \doteq x + 1)$. Then the expression $s_1 \in I \wedge s_2 \in S \wedge s_3 \in S \wedge (s_1, s_2) \in R \wedge (s_2, s_3) \in R$ (i.e., paths from initial states with length three) can be given to an SMT solver as follows: $x_1 \doteq 0 \wedge x_1 < 5 \wedge x_2 < 5 \wedge x_3 < 5 \wedge x_2 \doteq x_1 + 1 \wedge x_3 \doteq x_2 + 1$. Only the assignment $\{x_1 \mapsto 0, x_2, \mapsto 1, x_3 \mapsto 2\}$ satisfies the previous formula, which corresponds to the path $\pi = (x \doteq 0, x \doteq 1, x \doteq 2)$.*

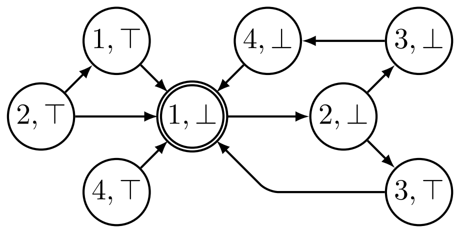

$$\begin{tikzcd}
 & (1,\top) \arrow[dr] & & (4,\bot) \arrow[dl] & & (3,\bot) \arrow[ll] \\
(2,\top) \arrow[ur] \arrow[rr] & & (1,\bot) \arrow[rr] & & (2,\bot) \arrow[ur] \arrow[dr] & \\
 & (4,\top) \arrow[ur] & & & & (3,\top) \arrow[ulll]
\end{tikzcd}$$

Figure 2.2: *Transition system corresponding to the symbolic transition system in Listing 2.1.*

2.3 Temporal logic

Propositional and first order logic can express static properties and behaviors of a system. However, when the state of a system changes over time dynamically, one may also want to reason about the *order* of the events. *Temporal logic* formalisms have the ability to capture the order of events with a logical time, i.e., without introducing time explicitly. Temporal logic formalisms are usually divided into two categories based on the structure of the logical time. Temporal logics with *linear time* can describe one linear sequence of events, while *branching time* temporal logics can describe all possible sequences.

Lamport described a simple linear time and a simple branching time logic in [27] and investigated their expressive power. He concluded that each logic can express some properties that the other cannot. Emerson and Halpern revisited this question in [28] and proposed a logic called CTL*, which combines linear and branching time logics. The linear fragment of CTL* is called *Linear Temporal Logic* (LTL), while the branching part is called *Computation Tree Logic* (CTL). In my work, I focus on a specific form of expressions, called safety properties. Safety properties are expressible in CTL*, CTL and LTL as well. Therefore, I give a short, informal introduction to CTL* (Section 2.3.1), CTL (Section 2.3.2), LTL (Section 2.3.3) and their expressive power (Section 2.3.4), but I only formalize safety properties (Section 2.3.5).

2.3.1 CTL*

CTL* is interpreted over a computation tree by extending propositional logic with *temporal operators* and *path quantifiers* [28].

The temporal operators are the following.

- $\mathsf{X}\ \varphi$ is the *next state operator*, which means that φ must hold in the next state of the path.
- $\mathsf{F}\ \varphi$ is the *future* (or eventually) *operator*, which means that φ must hold in at least one state along the path (which can be the first state as well).
- $\mathsf{G}\ \varphi$ is the *globally operator*, which means that φ must hold on every state along the path.
- $\varphi\ \mathsf{U}\ \psi$ is the *until operator*, which means that ψ must hold in a future state along the path and until then φ must hold in every state. It is also valid if ψ holds in the first state and φ never holds.

The path quantifiers are the following.

- $\mathsf{A}\ \varphi$ is the *universal quantifier*, which means that φ must hold for every path starting from the current state.
- $\mathsf{E}\ \varphi$ is the *existential quantifier*, which means that φ must hold for at least one path starting from the current state.

ACTL* is the fragment of CTL*, where only the universal quantifier (A) is used and negations are restricted to atomic formulas. Some algorithms, for example the abstraction framework defined in Section 4.1 only support ACTL* instead of the full CTL*. Analogously, in ECTL* only the existential quantifier (E) is used and negations are restricted to atomic formulas.

Example 2.14. *Some example requirements for the Kripke structure of the traffic light (Figure 2.1) are presented below.*

- EF *green: there is a behavior of the light, where it turns green.*
- AF *green: the light will always turn green eventually.*
- $\mathsf{AG}\ \neg(red \wedge green)$*: the light cannot be red and green at the same time.*

- *AG [off $\rightarrow$ AX (red $\vee$ off)]: if the light is off, then it can only be off or red in the next state.*
- *AGF off: the light can always turn off.*
- *E (red $\wedge$ X off): there is a behavior where the light starts from red but turns off in the next state.*

2.3.2 CTL

CTL is a fragment of CTL* with the following restriction. Valid operators must contain a single path quantifier followed by a temporal operator. Therefore, there are eight possible operators (besides the logical operators): $\mathsf{AX}, \mathsf{AF}, \mathsf{AG}, \mathsf{AU}, \mathsf{EX}, \mathsf{EF}, \mathsf{EG}, \mathsf{EU}$. The intuitive meaning of the operators is presented in Figure 2.3. States where φ and ψ hold are colored gray and black respectively.

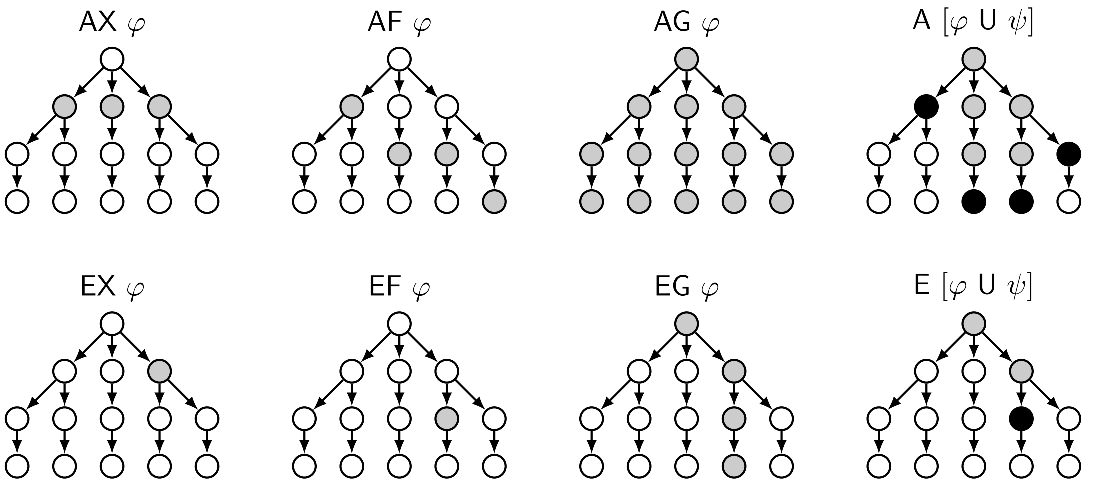


Figure 2.3: *Illustration of CTL operators.*

2.3.3 LTL

LTL is the linear time fragment of CTL*, i.e., LTL formulas are interpreted over a single path in the Kripke structure. Therefore, there are no path quantifiers in LTL. The temporal operators are equivalent to those in CTL*. An illustration of the operators can be seen in Figure 2.4. States where φ and ψ hold are colored gray and black respectively.

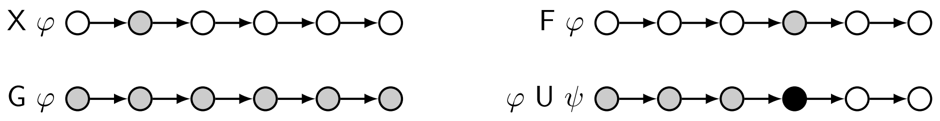


Figure 2.4: *Illustration of LTL operators.*

An LTL formula φ is valid for a Kripke structure M if it holds for all paths starting from the initial states. This also means that the corresponding CTL* formula of an LTL formula φ is A φ.

2.3.4 Expressive power of temporal logics

The expressive power of CTL and LTL is different due to the structure of the logical time they are interpreted on. LTL cannot express the possibility of an event (quantifier E), but only the necessity (quantifier A). On the other hand CTL cannot combine path operators. CTL* contains both LTL and CTL and also has expressions that are not expressible in any of them. Figure 2.5 presents some examples for each category.

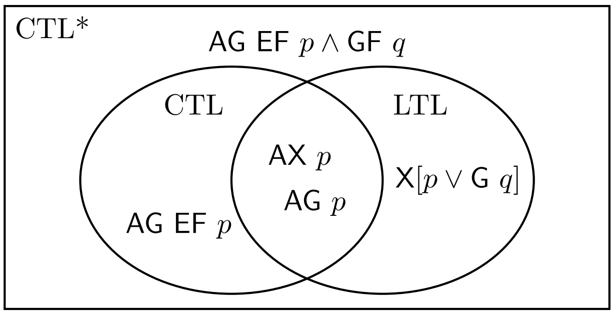

CTL*
$\mathsf{AG}\ \mathsf{EF}\ p \wedge \mathsf{GF}\ q$
CTL
LTL
$\mathsf{AX}\ p$
$\mathsf{AG}\ p$
$\mathsf{X}[p \vee \mathsf{G}\ q]$
$\mathsf{AG}\ \mathsf{EF}\ p$

Figure 2.5: *Comparison of the expressive power of temporal logics.*

2.3.5 Safety properties

In my work I focus on *safety properties* of the form $\mathsf{AG}\ \varphi$, where φ contains no temporal operators or path quantifiers. This is usually interpreted as φ is "something good" that must hold for every state along every path. Safety properties can also be thought of in the form $\mathsf{AG}\ \neg\psi$, where ψ is "something bad" that must never happen. It is clear that both representations are equivalent by choosing ψ to be $\neg\varphi$.

Given a Kripke structure M, a state $s \in S$ and a formula φ the notation $(M, s) \models \varphi$ means that φ holds in s in M. When M is clear from the context, I also denote $(M, s) \models \varphi$ simply by $s \models \varphi$. Suppose, that $p \in A$ is an atomic proposition and φ_1, φ_2 are propositional formulas. Then, the relation $\models$ for safety properties is defined recursively as follows:

1. $(M, s) \models p$ iff $p \in L(s)$,
2. $(M, s) \models \neg\varphi_1$ iff $(M, s) \not\models \varphi_1$,
3. $(M, s) \models \varphi_1 \wedge \varphi_2$ iff $(M, s) \models \varphi_1 \wedge (M, s) \models \varphi_2$,
4. $(M, s) \models \varphi_1 \vee \varphi_2$ iff $(M, s) \models \varphi_1 \vee (M, s) \models \varphi_2$,
5. $(M, s) \models \varphi_1 \rightarrow \varphi_2$ iff $(M, s) \models \varphi_1 \rightarrow (M, s) \models \varphi_2$,
6. $(M, s) \models \varphi_1 \leftrightarrow \varphi_2$ iff $(M, s) \models \varphi_1 \leftrightarrow (M, s) \models \varphi_2$,
7. $(M, s) \models \mathsf{AG}\ \varphi_1$ iff $(M, s_i) \models \varphi_1$ holds for each state $s_i \in \pi$ of each path $\pi = (s_0, s_1, \ldots)$ with $s_0 = s$.

2.4 Model checking

Model checking[2] is a formal verification technique aiming to automatically determine whether a (hardware or software) system meets a given requirement by analyzing all the possible behaviors. Model checking was first described by Clarke and Emerson [30] and independently by Queille and Sifakis [31]. Formally, the model checking problem is as follows [29].

Definition 2.8 (Model checking problem). Let M be a Kripke structure (i.e., state-transition graph). Let φ be a formula of temporal logic (i.e., the requirement). Find all states s of M such that $(M, s) \models \varphi$.

This means that the purpose of model checking is to find every state $s \in S$, which meets the requirement, i.e., the behavior of the model starting from s satisfies φ. However, the model checking problem is usually more specific as we are only interested in checking whether φ holds from the initial states, i.e., checking whether $(M, s_0) \models \varphi$ for all $s_0 \in I$.

Counterexamples. An important feature of safety properties of the form $\mathsf{AG}\ \varphi$ is that if the formula does not hold, a *counterexample* can be given, which is a (loop-free) path leading from an initial state to a state s with $(M, s) \not\models \varphi$, i.e., a state violating φ. There are also classes of formulas where counterexamples are loops or even tree-like structures [32].

State space generation. Models are usually given in high-level formalisms instead of Kripke structures. Since high-level formalisms also have precise syntax and semantics, these models can be translated into equivalent Kripke structures. This technique is called *state space generation* or *exploration*. However, high-level formalisms offer a compact representation of the model. Consequently, the size of the corresponding Kripke structure can be exponentially large (or even larger) compared to the size of the high-level model. This problem is referred to as the "state space explosion problem" in the literature. Therefore, explicitly enumerating and checking all the behaviors (or even states) of a high-level model is impossible in practice. Over the past decades, several types of advanced model checking algorithms were developed to overcome the state space explosion problem. Chapter 3 presents some of these techniques. For more details on model checking, the reader is referred to [33].

Example 2.15. *Consider a symbolic transition system $T = (V, Inv, Tran, Init)$, with*

- $V = \{v_1, v_2, \dots, v_n\}$ *and* $D_{v_i} = \mathbb{N}$ *for* $1 \le i \le n$,
- $Inv = \bigwedge_{i=1}^{n} (v_i \le k)$,
- $Tran = \bigwedge_{i=1}^{n} (v_i' \doteq v_i + 1) \vee (v_i' \doteq v_i)$,

[2]Clarke pointed out an interesting fact in [29]. Many people think that the term "model" in "model checking" refers to the formal representation of the system under verification. However, it originally intended to mean that the formula (corresponding to the requirement) holds for the Kripke structure (of the system), i.e., the Kripke structure is a *model* of the formula.

- $\mathit{Init} = \bigwedge_{i=1}^{n}(v_i \doteq 1)$.

For a given n and k, the symbolic transition system is described by n variables and $4n$ atoms. However, due to the domains and the invariant, the corresponding transition system (S, R, I) has $|S| = k^n$ states. The transition relation describes that each variable at each state stays the same or increments. This yields $|R|$ to be approximately $|S| \cdot 2^n$. It is only an approximation since if $v_i = k$, then it cannot be incremented.

Chapter 3

Related work

This chapter briefly summarizes the related work in the field of model checking. Section 3.1 presents some model checking techniques in general, while Section 3.2 focuses on CEGAR-based approaches and related techniques, which are also the main topic of my thesis.

3.1 Model checking approaches

This section first introduces explicit techniques in model checking along with their complexity (Section 3.1.1). However, explicit techniques are usually limited by the state space explosion problem. Therefore, the rest of the section briefly describes some of the advanced model checking approaches that were developed in the past decades, including partial order reduction (Section 3.1.2), symbolic methods (Section 3.1.3), bounded model checking (Section 3.1.4) and abstraction-based techniques (Section 3.1.5).

3.1.1 Explicit model checking

The most straightforward way to decide the model checking problem is to enumerate all states and paths from the initial states and check whether they satisfy the formula. Such techniques are called *explicit* methods.

Due to the restricted operators, CTL model checking can be performed by only analyzing the states of the Kripke structure instead of the paths [30]. At first, each state is labeled with the atomic propositions that hold on that state. Then, states are labeled with subformulas that hold on that state iteratively, until a fixpoint is reached. The complexity of the algorithm is thus, $O(|S| \cdot |\varphi|)$, where $|S|$ is the number of states in the Kripke structure M and $|\varphi|$ is the size of the requirement [34].

LTL model checking in contrast, is a computationally harder problem since paths of the Kripke structure have to be checked. This is often done by transforming the model checking problem into the emptiness checking of Büchi automata [35]. The complexity of the approach is $2^{O(|\varphi|)}O(|S|)$ [34]. The algorithms for LTL can be easily adopted to CTL*, hence the complexity of CTL* model checking is equivalent to LTL [34].

For safety properties of the form $\mathsf{AG}\ \varphi$, model checking reduces to *reachability analysis*, i.e., checking if a state s with $(M, s) \not\models \varphi$ can be reached from an initial state through transitions. This can be done in $O(|S| \cdot |\varphi|)$ complexity, using for example depth-first search to traverse S and check φ at each state. The problem however, is still state space explosion: the Kripke structure can be exponentially large compared to the high-level model.

3.1.2 Partial order reduction

State space explosion is often caused by the interleaving semantics of concurrent systems, since the local executions can interleave in many possible global orderings. However, the requirement φ is often insensitive to the ordering of some executions, e.g., between two synchronization points. *Partial order reduction* methods [36, 37] exploit this symmetry to reduce the size of the state space by partitioning the executions into equivalence classes. Each execution in the same equivalence class either satisfies or contradicts φ. Therefore, it is sufficient to choose only a single *representative* execution from each equivalence class, which yields a smaller state space.

3.1.3 Symbolic model checking

Symbolic model checking techniques [38] try to exploit regularities and symmetries in the model in order to have a more compact representation of the state space. States of the system are encoded by $\lceil \log_2 |S| \rceil$ Boolean variables. A set of states (e.g., initial states) $X \subseteq S$ is represented by a Boolean function $f_X \colon S \mapsto \{\top, \bot\}$, where $f_X(s) = \top$ if $s \in X$ and $f_x(s) = \bot$ if $s \notin X$. A set of state pairs (e.g., transition relation) can be represented with Boolean functions similarly. For example, if a system is represented by two Boolean variables x and y, and the set of reachable states is $\{(0,1),(1,0),(1,1)\}$, then this explicit list can be represented by the Boolean function $x \vee y$ in a compact way. Furthermore, Boolean functions can be efficiently encoded and manipulated using *reduced ordered binary decision diagrams* (ROBDDs) [39] or other types of decision diagrams.

3.1.4 Bounded model checking

Bounded model checking (BMC) techniques [40] exploit the fact that in many cases it is not necessary to explore the full state space in order to verify the requirement. For example, consider a safety property $\mathsf{AG}\ \varphi$, i.e., checking whether a state can be reached where φ does not hold. If such state can be found within a few steps, the rest of the state space does not have to be explored. Bounded model checking considers a k-bounded part of the state space, i.e., states that can be reached in k steps from an initial state.

However, it is not sure that the truth of an expression can be evaluated using a bound k. For example, if $\mathsf{AG}\ \varphi$ is true for a given k, it may fail for a $k' > k$ if a state violating φ can be reached in k' steps. Therefore, model checking has to be repeated, but with a larger k. On the other hand, for example if $\mathsf{AG}\ \neg\varphi$ holds for a bound k, then it also holds for $k' > k$. Bounded model checking can be efficiently transformed into a SAT problem [8]. The process is illustrated in Figure 3.1.

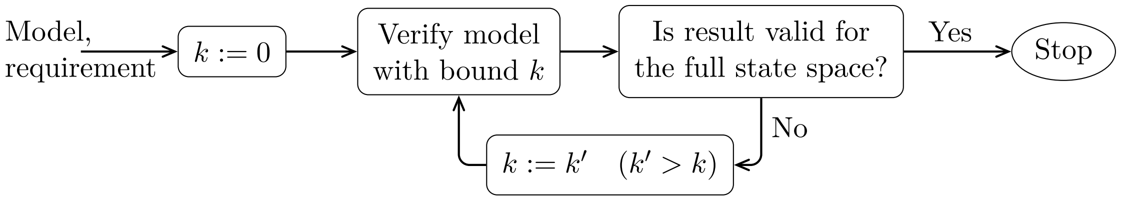

Model,
requirement
$k := 0$
Verify model with bound k
Is result valid for the full state space?
Yes
Stop
No
$k := k' \quad (k' > k)$

Figure 3.1: *Bounded model checking process.*

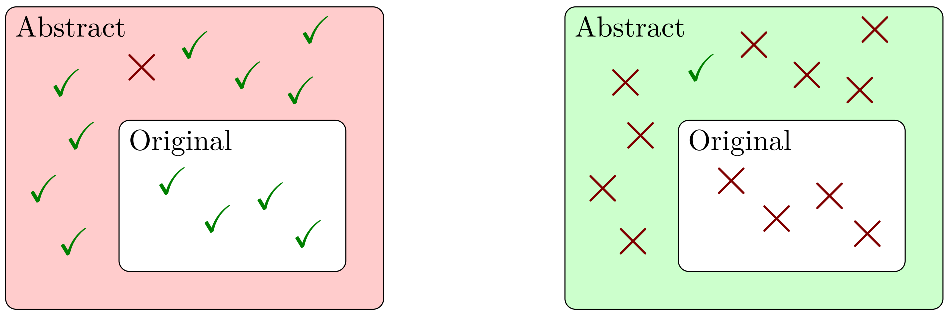
3.1.5 Abstraction-based model checking

Abstraction is a general mathematical approach, which is widely used to solve hard problems. It is also an important technique in model checking, where abstraction means to remove or simplify some details that are believed to be irrelevant for the requirement. Consequently, verifying the abstract model is computationally easier than the original one. However, the loss of information may lead to wrong results. *Abstraction-based* techniques are usually one-sided, which means that they accept either false negatives or false positives, but not both. Therefore, abstraction-based techniques are usually categorized by the way they control the information loss.

- *Over-approximation* techniques [41] relax constraints in the model, which extends the set of possible behaviors. Hence, if an ACTL* requirement φ holds in the abstract model, then it also holds for all behaviors of the original one. However, the new behaviors may introduce false negatives, i.e., counterexamples violating φ that are not present in the original model (Figure 3.2(a)). For ECTL* requirements ψ the opposite holds: if the abstract model violates ψ, then no behavior of the original model can be found that satisfies ψ. However, the new behaviors may introduce false positives, i.e., behaviors satisfying ψ that are only present in the abstract model (Figure 3.2(b)).

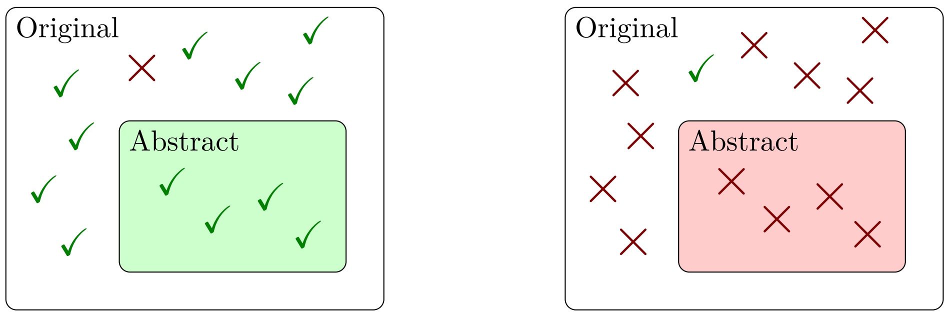
- *Under-approximation* [42] techniques remove behaviors from the model. Thus, if a behavior of the abstract model violates the ACTL* requirement φ, then it is also a counterexample in the original model. However, false positives may occur if the φ holds in the abstract model, but a behavior that is only present in the original model violates it (Figure 3.3(a)). For ECTL* requirements ψ the opposite holds: if a behavior satisfying ψ is present in the abstract model, then it is also included in the original one. However, reducing the set of behaviors may yield false negatives, i.e., a behavior satisfying ψ is lost in the abstract model (Figure 3.3(b)).

Note that the size of the rectangles in Figure 3.2 do not correlate with the complexity of model checking. Even though the abstract model over-approximates the behaviors of the original one, its representation of the state space can be smaller.



(a) False negative for ACTL*. (b) False positive for ECTL*.

Figure 3.2: *Illustration of over-approximation.*



(a) False positive for ACTL*. (b) False negative for ECTL*.

Figure 3.3: *Illustration of under-approximation.*

There are also techniques (e.g., abstract interpretation [43] or 3-valued logics [44]) that provide precise answers both in the positive and negative case, but may leave the problem undecided for some instances.

The main challenge in abstraction-based model checking is to find the right level of abstraction, which is coarse enough to avoid state space explosion but still fine enough to prove (or disprove) the desired requirement. *Counterexample-Guided Abstraction Refinement (CEGAR)* [45] is an automatic method for finding the proper level of abstraction. CEGAR-based approaches first start with a coarse abstraction and apply refinement based on counterexamples until the requirement is proved or disproved. My thesis also focuses on over-approximation techniques and CEGAR.

3.2 CEGAR-based model checking

CEGAR is a general concept that is widely used in formal verification. The work of Clarke et al. [45] (and its extended version [46]) was not the first verification method using counterexample-based abstraction refinement, however this approach is considered as the basis of the currently used CEGAR algorithms. This section briefly summarizes results in the field of CEGAR and related techniques. I describe some abstraction subtypes (Section 3.2.1), modeling formalisms (Section 3.2.2) and the most important related techniques that can be combined with CEGAR (Section 3.2.3). Finally, I present some of the tools that implement the CEGAR approaches and related concepts (Section 3.2.4).

3.2.1 Abstraction types

CEGAR can work with different types of abstractions. Predicate abstraction and partitioning variables into visible and invisible sets are widely used abstraction schemes, but there are other methods for the approximation of the state space.

Predicate abstraction. In *predicate abstraction* [46–49], states of the model are not tracked explicitly, but only their evaluation on a set of predicates $\mathcal{P}$. A predicate can either be satisfied or contradicted, therefore the abstract model has $2^{|\mathcal{P}|}$ states. Concrete states are mapped to abstract states based on the predicates they satisfy. For example, if $\mathcal{P} = \{x > y, y \doteq 2\}$, the concrete state $(x \doteq 3, y \doteq 1)$ belongs to the abstract state where $x > y$ holds, but $y \doteq 2$ does not. If the abstraction is too coarse, refinement is done by extending the set of predicates. New predicates are usually obtained by distinguishing the false negative from the other behaviors.

Visibility-based abstraction. Another widespread abstraction scheme is to partition the variables of the system into two sets: *visible* and *invisible* variables [50–52]. Invisible variables are considered to be irrelevant in verifying the system and thus, are abstracted out. Therefore, the state space of the abstract model is only defined by the visible variables. Usually, visible variables are those that appear in the requirement. If the abstraction is too coarse, refinement is obtained by making some of the previously invisible variables visible. New variables are usually those that can distinguish the false negative from the other behaviors.

Other methods. There are also other methods where abstraction amounts to the over-approximation of (reachable) states with *interpolants* [53] or *inductive invariants* [54]. Inductive invariants are similar to the induction concept in mathematical proofs. A formula φ is an inductive invariant if (1) it holds for the initial states ($s_0 \models \varphi$ for each $s_0 \in I$) and (2) if it holds for a state, it also holds for its successors ($s \models \varphi \Rightarrow s' \models \varphi$ for each s' with $(s, s') \in R$). However, induction in one step is usually too strict. Therefore, k-induction is often used, which is the generalization of induction from one step to paths of length k. If the requirement cannot be verified on the over-approximated model, the interpolant or the inductive invariant is refined.

3.2.2 Modeling formalisms

Clarke et al. originally described CEGAR for hardware model checking, where the models are usually transition systems or Kripke structures [46, 50]. Since then, CEGAR has been successfully applied in software model checking, i.e., to the verification of programs [48,49,51,55]. Programs are described by a *control flow automaton (CFA)*, which consists of program locations and control flow edges. Locations describe the actual state of the program counter, while edges describe the operations executed on variables when control flows from one location to another. A state of the program is thus, defined by the location and the evaluation of the variables. Beyer and Löwe [51] achieve abstraction by making some variables invisible at each location. Ermis et al. [48] and Leucker et al. [49] use predicate abstraction over the variables: initially each location is assigned with a "true" predicate, which is then refined based on counterexamples. The abstraction of McMillan [55] lies in the partial unwinding of the control flow graph.

CEGAR can also be applied to the verification of *hybrid systems* [56]. Hybrid systems combine discrete and continuous state variables. Models usually have an infinite state space, where the behavior is defined by differential equations in each discrete state. In order to be able to perform model checking, a finite abstraction is produced and refined using CEGAR.

In my former work I was developing a CEGAR-based algorithm for the reachability analysis of *Petri nets* [57]. Petri nets are graphical models for describing and analyzing parallel, asynchronous and distributed systems. The behavior of a Petri net is determined by the set of reachable states. The so-called *state equation* of Petri nets is a set of linear inequalities that over-approximate reachable states. Hence, it can be used as an abstraction. The power of state equation lies in the fact that it is only based on the static structure of the Petri net. However, false negatives can occur that are eliminated by extending the state equation with additional linear inequalities.

Gmeiner et al. [58] used CEGAR for the *parameterized verification* of fault-tolerant distributed algorithms. Given a concrete value k, a distributed algorithm can be verified for k participants using traditional model checking approaches. Parameterized verification means to check the algorithm for all values of k at once, which requires abstraction due to the large (or infinite) number of possible values for k. Gmeiner et al. proposed a method where the potentially infinite number of participants are divided into a finite number of intervals, based on guards in the algorithm.

3.2.3 Combining with other techniques

CEGAR can work with different types of model checkers to verify the abstract model. The only requirement is that the model checker should be able to give a counterexample if the result is negative. Clarke et al. used both symbolic [46] and SAT-based [50] algorithms. However, explicit methods can also be used when the abstraction is coarse [51].

There are also other techniques that can improve the efficiency of CEGAR-based algorithms. *Lazy abstraction* is an approach that refines only a subset of the abstract state space. It was first described by Henzinger et al. [59] for the verification of programs. In their approach, the state space is constructed on-the-fly and the newly introduced predicate is only considered in the unexplored part of the state space (and also a subset of the explored part, due to loops). Ermis et al. [48] achieve lazy abstraction by only refining program locations with the newly introduced predicate that are part of the counterexample. Beyer and Löwe [51] use invisible variables, where lazy abstraction means to make variables visible only at program locations appearing in the counterexample. Furthermore, a different set of variables can be made visible at different program locations.

Slicing is a technique introduced by Weiser to reduce a program to a minimal form that still produces the same behavior for a given slicing criterion [60]. It was first used in debugging, testing and maintenance to extract the relevant parts of a program. Since then, slicing was also introduced in model checking as a preprocessing step to reduce the state space. Brückner et al. [61] define different types of slicing operations to eliminate or simplify abstract states and transitions in the abstract state space of their CEGAR approach.

Interpolation is often used to infer new predicates that refine the abstraction. Heinzinger et al. [62] and Brückner et al. [61] use Craig interpolation to refine a single state that causes the false negative. In contrast, Beyer et al. [51], Ermis et al. [48] and McMillan [55] use sequence interpolation to refine multiple states along the counterexample. Furthermore, Beyer et al. [63] recently proposed a technique that considers multiple prefixes of the counterexample to generate alternative refinements. Consequently, refinements can be ordered according to some metric (e.g., coarseness) and the "best" one can be selected.

New predicates can also be obtained by *unsat cores*. The unsat core of an unsatisfiable CNF formula is a subset of clauses that are already unsatisfiable. Leucker et al. [49] encode the counterexample into an unsatisfiable formula and use minimal unsat cores to eliminate clauses that have no impact on the counterexample being a false negative. The advantage of their approach is that it can yield smaller predicates than interpolation and it can also be used if the underlying SMT solver does not support interpolation.

CEGAR can also be combined with induction. A major drawback of k-induction is that the second condition (the induction part) is usually too general because it also corresponds to paths that cannot be reached. Beyer et al. [54] use a CEGAR-based algorithm in parallel with k-induction to explore an abstract state space and to strengthen the invariants.

3.2.4 Tools

There are several existing tools that implement CEGAR algorithms along with the related techniques. Clarke et al. implemented their approaches [46,50] in the NuSMV tool, which is the state-of-the-art reimplementation of the symbolic model checker SMV [64]. Beyer et al. [51, 63] presented their work within the CPAChecker framework, which is a tool for configurable software verification. The tool of Ermis et al. [48] is called ULTIMATE and is implemented as a chain of Eclipse plug-ins. The lazy abstraction concept was first introduced in the BLAST tool of Henzinger et al. [59]. The SLAB tool of Brücker et al. [61] also utilizes the slicing technique besides lazy abstraction.

Chapter 4

Counterexample-Guided Abstraction Refinement

This chapter presents CEGAR, the main topic of my work. CEGAR aims to solve a main challenge in abstraction-based model checking, namely finding the proper level of abstraction. It is a general concept that can work with different subtypes of abstraction. First, I describe a generic CEGAR framework for existential abstraction (Section 4.1), in which the presented algorithms fit. Then I present two approaches that are mainly based on existing algorithms from the literature: *clustered CEGAR* (Section 4.2) and *visibility-based CEGAR* (Section 4.3). In Section 4.4 I propose a new algorithm, called the *interpolating CEGAR*, which is a combination of the clustered and visibility-based approaches and the related techniques (presented in Section 3.2). Finally, in Section 4.5, I summarize the common and different aspects of the three CEGAR approaches presented in this chapter and I also emphasize my contributions.

4.1 A generic CEGAR framework

In this section I describe a generic framework for existential abstraction based on [46]. First, I introduce abstraction functions and their important properties (Section 4.1.1), then I present the main steps of a generic CEGAR approach (Section 4.1.2).

4.1.1 Existential abstraction

Existential abstraction simplifies the model by systematically relaxing constraints [41]. This yields an over-approximation in the possible behaviors: the abstract model keeps all behaviors of the original one, but may add new ones. Existential abstraction is usually achieved by partitioning the concrete states into disjoint groups, each group being an abstract state [46].

Definition 4.1 (Abstraction function). Given a Kripke structure $M = (S, R, L, I)$, *abstraction* is a function $\hbar \colon S \mapsto \hat{S}$, where $\hat{S}$ denotes the set of abstract states. An abstract state $\hat{s} \in \hat{S}$ *abstracts* a concrete state $s \in S$ if $\hbar(s) = \hat{s}$.

Abstraction can also be viewed as an *equivalence relation* (denoted by $\equiv_\hbar$) over the concrete states S, with the abstract states being the *equivalence classes*. Thus, for two concrete states $s_1, s_2 \in S$, $\hbar(s_1) = \hbar(s_2)$ can also be denoted by $s_1 \equiv_\hbar s_2$ or $(s_1, s_2) \in \equiv_\hbar$. In the opposite direction, $\hbar^{-1}$ is defined in the following way.

Definition 4.2 (Concretization function). Given an abstraction function $\hbar \colon S \mapsto \hat{S}$, the *concretization function* $\hbar^{-1} \colon \hat{S} \mapsto 2^S$, is given by $\hbar^{-1}(\hat{s}) = \{s \in S \mid \hbar(s) = \hat{s}\}$ for each abstract state $\hat{s} \in \hat{S}$.

In other words, the concretization of an abstract state $\hat{s} \in \hat{S}$ is the set of concrete states that are mapped to $\hat{s}$. The abstract Kripke structure generated by $\hbar$ is as follows.

Definition 4.3 (Abstract Kripke structure). Given a (concrete) Kripke structure $M = (S, R, L, I)$ and an abstraction function $\hbar \colon S \mapsto \hat{S}$, the *abstract Kripke structure* $\hat{M} = (\hat{S}, \hat{R}, \hat{L}, \hat{I})$ is defined in the following way.

- $(\hat{s}, \hat{s}') \in \hat{R}$ iff $\exists (s, s') \in R$ with $\hbar(s) = \hat{s}$ and $\hbar(s') = \hat{s}'$,
- $\hat{s} \in \hat{I}$ iff $\exists s \in I$ with $\hbar(s) = \hat{s}$,
- $\hat{L}(\hat{s}) = \bigcup_{s \in \hbar^{-1}(\hat{s})} L(s)$.

Informally, the set of abstract states is created by mapping multiple concrete states to single abstract states. A transition exists between two abstract states if there is at least one transition between the abstracted concrete states. An abstract state is initial if it abstracts at least one concrete initial state. The labels of an abstract state are obtained by taking the union of the labels of the abstracted concrete states.

To distinguish between concrete and abstract states, I denote them in figures with circles and rectangles respectively. Furthermore, when both abstract and concrete states are present in a figure, I denote $\hbar(s) = \hat{s}$ by drawing s inside $\hat{s}$.

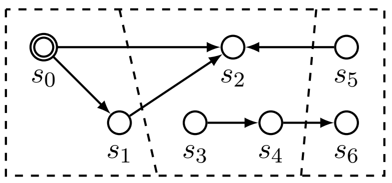
Example 4.1 (from [46]). *Consider the concrete Kripke structure in Figure 4.1(a). Suppose that the states are grouped by $\hbar$ into three abstract states as indicated by the dashed lines. The corresponding abstract Kripke structure can be seen in Figure 4.1(b). The states are mapped in the following way:*

- $\hbar(s_0) = \hbar(s_1) = \hat{s}_0$,
- $\hbar(s_2) = \hbar(s_3) = \hbar(s_4) = \hat{s}_1$,
- $\hbar(s_5) = \hbar(s_6) = \hat{s}_2$.

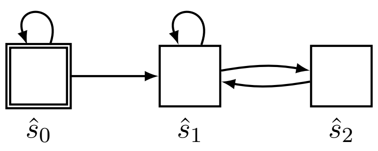
The transitions are mapped in the following way:

- $(s_0, s_1) \in R \mapsto (\hat{s}_0, \hat{s}_0) \in \hat{R}$,
- $(s_0, s_2) \in R \mapsto (\hat{s}_0, \hat{s}_1) \in \hat{R}$,
- $(s_1, s_2) \in R \mapsto (\hat{s}_0, \hat{s}_1) \in \hat{R}$,
- $(s_3, s_4) \in R \mapsto (\hat{s}_1, \hat{s}_1) \in \hat{R}$,
- $(s_4, s_6) \in R \mapsto (\hat{s}_1, \hat{s}_2) \in \hat{R}$,
- $(s_5, s_2) \in R \mapsto (\hat{s}_2, \hat{s}_1) \in \hat{R}$.

Furthermore, only $\hat{s}_0 \in \hat{I}$, since $I = \{s_0\}$ and $\hbar(s_0) = \hat{s}_0$.



(a) Concrete Kripke structure.



(b) Abstract Kripke structure.

Figure 4.1: *Existential abstraction example.*

In general, there are $|\hat{S}|^{|S|}$ possible mappings $\hbar$ from S to $\hat{S}$. However, in order to be able to reason about a temporal logic formula on the abstract model, $\hbar$ must follow some rules.

Definition 4.4 (Appropriateness). An abstraction function $\hbar$ is *appropriate* for an ACTL* formula φ if for all atomic subformulas φ_0 of φ and for all states $s_1, s_2 \in S$ with $s_1 \equiv_\hbar s_2$ it holds that $s_1 \models \varphi_0 \Leftrightarrow s_2 \models \varphi_0$.

Informally, $\hbar$ is appropriate for φ if concrete states within the same abstract state cannot be distinguished by atomic subformulas of φ.

Theorem 1 (from [46]). If $\hbar$ is appropriate for the formula φ and M is labeled with the atomic propositions in φ, then $\hat{M} \models \varphi \Rightarrow M \models \varphi$.

Theorem 1 ensures that if the abstraction function is appropriate, then no false positives are accepted, i.e., the abstraction is an over-approximation. On the other hand, false negatives may occur using existential abstraction. Even though the abstract model violates the requirement, the concrete one may still satisfy it. In my work I focus on safety properties of the form $\mathsf{AG}\ \varphi$, for which counterexamples are loop-free paths. The definition of $\hbar^{-1}$ can be extended to abstract paths in the following way.

Definition 4.5 (Path concretization). Given an abstract path $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n)$ with $\hat{s}_1 \in \hat{I}$, concrete paths corresponding to $\hat{\pi}$ are the following [46].

$$\hbar^{-1}(\hat{\pi}) = \left\{ (s_1, s_2, \ldots, s_n) \;\middle|\; s_1 \in I \wedge \bigwedge_{1 \le i \le n} \hbar(s_i) = \hat{s}_i \wedge \bigwedge_{1 \le i < n} (s_i, s_{i+1}) \in R \right\}$$

The path $\hat{\pi}$ is said to be *concretizable* if $\hbar^{-1}(\hat{\pi}) \neq \emptyset$.

Informally, the set $\hbar^{-1}(\hat{\pi})$ contains concrete paths where the first state is an initial state and the ith concrete state is mapped to the ith abstract state.

Example 4.2 (from [46]). *Recall the example in Figure 4.1 and suppose that s_5 and s_6 should not be reached because something "bad" happens there. In the abstract model therefore, the abstract state $\hat{s}_2$ should not be reached. An (abstract) counterexample is thus the path $\hat{\pi} = (\hat{s}_0, \hat{s}_1, \hat{s}_2)$. It is easy to see however, that $\hat{\pi}$ is not concretizable. Whichever concrete path is followed from the initial state, it is stuck at s_2, which is therefore, called a* dead-end state*. On the other hand s_4 is the state that has an outgoing edge to a state in $\hat{s}_2$, making the model checker believe that $\hat{s}_2$ can be reached. Thus, s_4 is called a* bad state*. The state s_3 is neither dead-end, nor bad, therefore it is called* irrelevant*.*

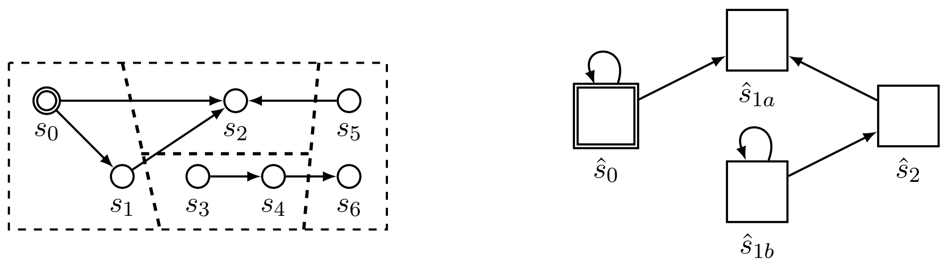
Abstraction refinement

When an abstract counterexample has no corresponding concrete counterexample, it is called *spurious*. Spurious counterexamples are caused by the coarseness of the abstraction and can be eliminated by refinement [46].

Definition 4.6 (Refinement). An abstraction function $\hbar' \colon S \mapsto \hat{S}'$ is a *refinement* of $\hbar \colon S \mapsto \hat{S}$ if $\equiv_{\hbar'} \subset \equiv_{\hbar}$ holds. In other words, $\hbar'(s_1) = \hbar'(s_2) \Rightarrow \hbar(s_1) = \hbar(s_2)$ for all $s_1, s_2 \in S$ and there exists $s_3, s_4 \in S$ for which $\hbar(s_3) = \hbar(s_4)$ but $\hbar'(s_3) \neq \hbar'(s_4)$.

Informally, $\hbar'$ is a refinement of $\hbar$ if equivalent states in $\hbar'$ are also equivalent in $\hbar$. However, some states that were equivalent in $\hbar$ are no longer equivalent in $\hbar'$, which means that some abstract states of $\hbar$ are split into multiple abstract states in $\hbar'$.

Example 4.3. *The spurious counterexample in Example 4.2 can be eliminated by separating the dead-end and bad states in $\hat{s}_1$. One possible refinement $\hbar'$ is illustrated in Figure 4.2(a) with the corresponding abstract Kripke structure in Figure 4.2(b). It is easy to see that $\hat{s}_2$ can no longer be reached, thus the spurious counterexample is eliminated.*

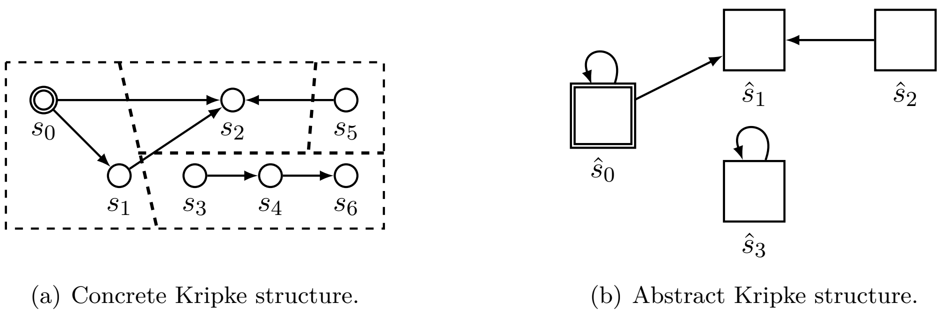


(a) Concrete Kripke structure. (b) Abstract Kripke structure.

Figure 4.2: *Abstraction refinement example.*

Example 4.4. *The abstraction $\hbar''$ in Figure 4.3 also eliminates the spurious counterexample, but it is not a refinement of $\hbar$ (Figure 4.1). For example, $\hbar''(s_4) = \hbar''(s_6)$ but $\hbar(s_4) \neq \hbar(s_6)$.*

In the previous examples abstraction was defined by explicitly mapping each concrete state to an abstract state. This requires the enumeration of all concrete states, which can lead



(a) Concrete Kripke structure. (b) Abstract Kripke structure.

Figure 4.3: *Abstraction eliminating the counterexample, but not being a refinement.*

to state space explosion. To tackle this problem, the concrete state space is usually given in an implicit, high-level representation and abstraction is also defined implicitly [46, 50]. Hence, the abstract state space can be constructed without enumerating the concrete states and transitions explicitly. In my work I define abstractions implicitly on symbolic transition systems, similarly to Clarke et al. [46] (see Section 4.2.1, Section 4.3.1 and Section 4.4.1).

4.1.2 The CEGAR loop

The previous section concluded that existential abstraction admits no false positives and when a false negative (spurious counterexample) occurs, it can be eliminated by refining the abstraction. However, it is possible that multiple refinements are required to eliminate all spurious counterexamples. Algorithms therefore, usually start with a coarse abstraction to keep the state space small and refine the abstraction iteratively until the proper level is reached, which is fine enough to prove or disprove the given requirement. This approach is called Counterexample-Guided Abstraction Refinement [46]. The main steps are illustrated in Figure 4.4 and explained below.

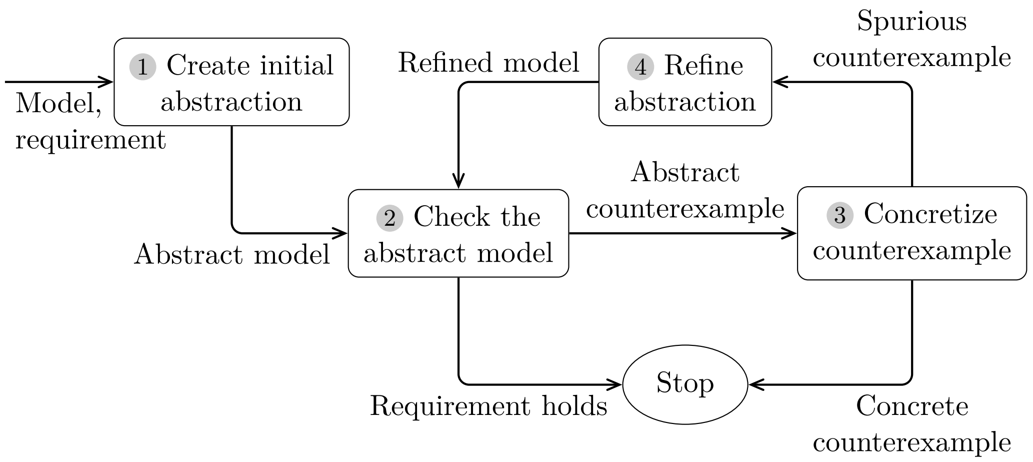


Figure 4.4: *Generic CEGAR process.*

1. The input of the algorithm is the concrete model (usually in a high-level representa-

tion) and the requirement (e.g., a temporal formula). The first step is to create an initial, usually coarse abstract model.

2. The abstract model is then checked by a model checking algorithm. The CEGAR approach can be combined with any type of model checker as long as it is capable of providing a counterexample. Due to over-approximation, if the abstract model satisfies the requirement, then it also holds in the concrete model.

3. On the other hand, if the abstract model violates the requirement, an abstract counterexample is produced by the model checker. The third step[1] is to find a concrete counterexample corresponding to the abstract one. This is usually done by exploring the subset of the concrete state space that corresponds to the abstract counterexample. If a concrete counterexample exists, it is a witness that the original model also violates the requirement.

4. If the abstract counterexample is found to be spurious, the abstraction has to be refined and the process has to be repeated from Step 2, until either the requirement holds for the abstract model or a concrete counterexample is found.

If the original model has a finite number of states, then the initial equivalence relation $\equiv_{\hbar_0}$ corresponding to the initial abstraction $\hbar_0$ is also finite. In the ith refinement iteration $\equiv_{\hbar_{i+1}} \subset \equiv_{\hbar_i}$ holds, thus a finite k must exist where $\equiv_{\hbar_k} = \emptyset$. Then, refinement is no longer possible and the algorithm terminates. Informally, $\equiv_{\hbar_k} = \emptyset$ corresponds to an abstraction, where each concrete state belongs to a different abstract state, i.e., no spurious behavior is possible.

The main advantage of the CEGAR approach is therefore, that it is fully automatic. The user of the algorithm only needs to give the concrete model and the requirement. The disadvantage is that CEGAR-based methods usually use heuristics for the refinement, hence they may not produce an abstraction as coarse as algorithms that are guided with the help of the user (who has domain specific knowledge).

4.2 Clustered CEGAR

Clustered CEGAR is mainly based on the work of Clarke et al. [46]. The key idea of the approach is to group the variables of the model into clusters based on dependencies between them. Initial abstraction is then calculated separately for the clusters using composite abstraction functions and predicate abstraction (Section 4.2.1). The abstract model is checked by taking the product of the clusters on-the-fly (Section 4.2.2). An abstract counterexample is concretized by unfolding a part of the concrete model (Section 4.2.3). Refinement is achieved by splitting abstract states based on dead-end and bad states (Section 4.2.4).

[1]This step is sometimes called “examination” instead of “concretization” in the literature.

4.2.1 Initial abstraction

Given a symbolic transition system T with a finite set of variables $V = \{v_1, v_2, \dots, v_n\}$ with domains $D_{v_1}, D_{v_2}, \dots, D_{v_n}$ and a requirement φ, the concrete Kripke structure $M = (S, R, L, I)$ can be built in the following way. The set of states S, the transition relation R and the initial states I can be built as described in Section 2.2.2. Let $\mathsf{atoms}(T)$ be the set of atomic formulas that appear in the conditions[2] of the transitions or in the requirement. Then the labeling $L \colon S \mapsto 2^{\mathsf{atoms}(T)}$ is given by $L(s) = \{\varphi_0 \in \mathsf{atoms}(T) \mid s \models \varphi_0\}$, i.e., states are labeled with the atomic formulas that hold for them.

Composite abstraction functions. As argued in [46], it is usually computationally too expensive to calculate abstraction directly (i.e., building $\hat{M}$ directly from M) because M can be large. However, the state space S of the system is a (subset of the) product $D = D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$. It is therefore, a considerable idea to define abstraction functions $\hbar_i \colon D_{v_i} \mapsto \hat{D}_{v_i}$ separately for each domain. Consequently, given a state $d = (d_1, d_2, \dots, d_n)$, $\hbar(d)$ is equal to $(\hbar_1(d_1), \hbar_2(d_2), \dots, \hbar_n(d_n))$ and $\hat{S}$ equals to $\hat{D}_{v_1} \times \hat{D}_{v_2} \times \ldots \times \hat{D}_{v_n}$. Such $\hbar$ is called a *composite abstraction function*. However, the full power of abstraction cannot be exploited by defining abstraction separately for each domain and more spurious behavior can occur due to the dependencies between variables [46].

Example 4.5 (from [46]). *Let* $D = \{0,1,2\} \times \{0,1,2\}$ *and* $\hat{D} = \{0,1\} \times \{0,1\}$*. There are* $4^9 = 262\,144$ *possible mappings* $\hbar \colon D \mapsto \hat{D}$*. However, there are only* $2^3 = 8$ *mappings from* $\{0,1,2\}$ *to* $\{0,1\}$*, therefore* $8 \cdot 8 = 64$ *mappings from* D *to* $\hat{D}$ *if* $\hbar = (\hbar_1, \hbar_2)$*.*

Clarke et al. therefore, proposed a different approach [46]. Suppose that the system T has n variables $V = \{v_1, v_2, \dots, v_n\}$. The set V is partitioned into *variable clusters* $VC_1, VC_2, \dots, VC_m$ with $VC_1 \cup VC_2 \cup \ldots \cup VC_m = V$ and $VC_i \cap VC_j = \emptyset$ for $1 \leq i < j \leq m$. Each cluster VC_i has the domain $D_{VC_i} = \prod_{v \in VC_i} D_v$, thus $D = D_{VC_1} \times D_{VC_2} \times \ldots \times D_{VC_m}$. Let $\mathsf{var}(\varphi)$ denote the variables appearing in the formula φ, e.g., $\mathsf{var}(x < y + 4) = \{x, y\}$. The equivalence relation $\equiv_V$ clustering V is defined in the following way. Given two variables $v_1, v_2 \in V$,

$$v_1 \equiv_V v_2 \text{ iff } \exists \varphi \in \mathsf{atoms}(T) \text{ with } v_1, v_2 \in \mathsf{var}(\varphi).$$

Informally, two variables belong to the same cluster if they appear together in an atomic formula. Variable clusters can be calculated using the disjoint-set data structure [65]. Initially each variable belongs to a different cluster. Then each formula $\varphi \in \mathsf{atoms}(T)$ is checked and the clusters of the variables that appear in φ are joined.

Each variable cluster VC_i has an associated *formula cluster* $FC_i = \{\varphi \in \mathsf{atoms}(T) \mid \mathsf{var}(\varphi) \subseteq VC_i\}$, i.e., formulas that contain variables of the cluster. The component $\hbar_i \colon D_{VC_i} \mapsto \hat{D}_{VC_i}$ of the composite abstraction function $\hbar(\hbar_1, \hbar_2, \dots, \hbar_m)$ is defined on the

[2]The condition in the formula `if` φ_1 `then` φ_2 `else` φ_3 is φ_1.

domain D_{VC_i} in the following way, where $(d_1, d_2, \ldots, d_k), (e_1, e_2, \ldots, e_k) \in \prod_{v \in VC_i} D_v$ [46].

$$(d_1, d_2, \ldots, d_k) \equiv_{\hbar_i} (e_1, e_2, \ldots, e_k) \text{ iff } \bigwedge_{\varphi \in FC_i} (d_1, d_2, \ldots, d_k) \models \varphi \Leftrightarrow (e_1, e_2, \ldots, e_k) \models \varphi$$

Informally, this means that two states of the variable cluster are in the same equivalence class (i.e., abstract state) if they cannot be distinguished by the atomic formulas appearing in the formula cluster. Defining the abstraction function this way naturally ensures appropriateness since the atomic formulas contain the atoms of the requirement.

Since $S = D_{VC_1} \times D_{VC_2} \times \ldots \times D_{VC_m}$, a concrete state s can also be denoted by its components $s = (d_1, d_2, \ldots, d_m)$, where $d_i \in D_{VC_i}$. The abstraction of s is then defined by $\hbar(s) = (\hbar_1(d_1), \hbar_2(d_2), \ldots, \hbar_m(d_m))$. The abstract state space is given by the composition of the clusters, i.e., $\hat{S} = \hat{D}_{VC_1} \times \hat{D}_{VC_2} \times \ldots \times \hat{D}_{VC_m}$. Thus, an abstract state $\hat{s} \in \hat{S}$ can also be denoted by its components: $\hat{s} = (\hat{s}^1, \hat{s}^2, \ldots, \hat{s}^m)$, where $\hat{s}^i \in \hat{D}_{VC_i}$.

Example 4.6 (from [46]). *Consider the system T in Listing 4.1. The system has three variables $V = \{x, y, reset\}$ with $D_x = D_y = \{0, 1, 2\}$ and $D_{reset} = \{0, 1\}$. The set of atomic formulas is $\mathsf{atoms}(T) = \{(reset \doteq 1), (x < y), (x \doteq y), (y \doteq 2)\}$. Therefore, there are two variable clusters $VC_1 = \{x, y\}$ and $VC_2 = \{reset\}$ with two formula clusters $FC_1 = \{(x < y), (x \doteq y), (y \doteq 2)\}$ and $FC_2 = \{(reset \doteq 1)\}$. Consider the variable cluster VC_1. There are $|D_x| \cdot |D_y| = 3 \cdot 3$ states in D_{VC_1}, which are represented by the cells in Table 4.1.*

Each cell contains the atomic formulas that hold for that state. It can be seen that for example $(0, 0)$ and $(1, 1)$ can belong to the same equivalence class because the same formulas (only $x \doteq y$) hold for them. The domain D_{VC_1} is partitioned into the following five equivalence classes (i.e., abstract states):

- $\hat{s}^1_0$: $(0, 0) \equiv_{\hbar_1} (1, 1)$,
- $\hat{s}^1_1$: $(1, 0) \equiv_{\hbar_1} (2, 0) \equiv_{\hbar_1} (2, 1)$,
- $\hat{s}^1_2$: $(0, 1)$,
- $\hat{s}^1_3$: $(0, 2) \equiv_{\hbar_1} (1, 2)$,
- $\hat{s}^1_4$: $(2, 2)$.

It can be shown similarly that the variable cluster VC_2 has two abstract states: $\hat{s}^2_0$ corresponding to $reset \doteq 0$, $\hat{s}^2_1$ corresponding to $reset \doteq 1$, and $\hat{s}^2_1$ is labeled with $(reset \doteq 1)$. The abstraction function $\hbar = (\hbar_1, \hbar_2)$ is thus defined by $\hbar_1 \colon \{0, 1, 2\}^2 \mapsto \{\hat{s}^1_0, \hat{s}^1_1, \hat{s}^1_2, \hat{s}^1_3, \hat{s}^1_4\}$ and $\hbar_2 \colon \{0, 1\} \mapsto \{\hat{s}^2_0, \hat{s}^2_1\}$, with $\hbar_1(0, 0) = \hbar_1(1, 1) = \hat{s}^1_0$, $\hbar_1(1, 0) = \hbar_1(2, 0) = \hbar_1(2, 1) = \hat{s}^1_1$, $\hbar_1(0, 1) = \hat{s}^1_2$, $\hbar_1(0, 2) = \hbar_1(1, 2) = \hat{s}^1_3$, $\hbar_1(2, 2) = \hat{s}^1_4$, $\hbar_2(0) = \hat{s}^2_0$ and $\hbar_2(1) = \hat{s}^2_1$. The abstract state space (i.e., the composition of the clusters) has the following $5 \cdot 2 = 10$ states: $(\hat{s}^1_0, \hat{s}^2_0)$, $(\hat{s}^1_1, \hat{s}^2_0)$, $(\hat{s}^1_2, \hat{s}^2_0)$, $(\hat{s}^1_3, \hat{s}^2_0)$, $(\hat{s}^1_4, \hat{s}^2_0)$, $(\hat{s}^1_0, \hat{s}^2_1)$, $(\hat{s}^1_1, \hat{s}^2_1)$, $(\hat{s}^1_2, \hat{s}^2_1)$, $(\hat{s}^1_3, \hat{s}^2_1)$, $(\hat{s}^1_4, \hat{s}^2_1)$.

Predicate abstraction. The example above shows that the abstract states and labeling can be constructed by enumerating and grouping the concrete states into abstract states.

Listing 4.1: *Example symbolic transition system described in the TTMC framework.*

```
specification System {
    property safe : {
        local var reset : integer
        local var x : integer
        local var y : integer

        invariant 0 <= reset and reset <= 1
        invariant 0 <= x and x <= 2
        invariant 0 <= y and y <= 2

        initial reset = 0
        initial x = 0
        initial y = 1

        transition reset' >= 0 and reset' <= 1

        transition x' = (
            if reset = 1 then 0
            else if x < y then x + 1
            else if x = y then 0
            else x
        )

        transition y' = (
            if reset = 1 then 0
            else if x = y and not y = 2 then y + 1
            else if x = y then 0
            else y
        )
    } models G(x<y or reset=1)
}
```

Table 4.1: *Example labeling of a cluster.*

	$x \doteq 0$	$x \doteq 1$	$x \doteq 2$
$y \doteq 0$	$(x \doteq y)$		
$y \doteq 1$	$(x < y)$	$(x \doteq y)$	
$y \doteq 2$	$(x < y), (y \doteq 2)$	$(x < y), (y \doteq 2)$	$(x \doteq y), (y \doteq 2)$

However, if a variable cluster contains many variables with large (or infinite) domains, this can be computationally expensive (or impossible).

Therefore, I adopt a different strategy, namely *predicate abstraction* [47] that only enumerates the abstract states. The idea behind predicate abstraction is that each abstract state either satisfies or contradicts an atomic formula, thus given k formulas there are 2^k possible abstract states. These can be enumerated using a tree where nodes contain formulas. The root node is empty and each node has two successors by extending the list with the next formula and its negation. Formally, let $FC_j = \{\varphi_1, \varphi_2, \dots, \varphi_k\}$ be the atomic formulas of the jth cluster. Consequently, the tree has $k + 1$ levels. The root node $N_1 = \emptyset$ is an empty list of formulas and the node N_i at level i has two successors: $N^1_{i+1} = N_i \cup \{\varphi_i\}$ and $N^2_{i+1} = N_i \cup \{\neg\varphi_i\}$. Each leaf node N^i_{k+1} on the level $k + 1$ is a possible abstract state (of the cluster). However, it has to be checked with an SMT solver that the formulas do not contradict each other and the invariant *Inv*. If a node is contradicting, it can be removed from the state space similarly to *slicing* [61]. This check can be done at inner nodes as well and if an inner node is already contradicting then the subtree under that node does not need to be calculated. The ideas above are formulated in Algorithm 1.

Algorithm 1: Enumerate abstract states with labels for a cluster.

```
    Input  : $FC_j = \{\varphi_1, \varphi_2, \dots, \varphi_k\}$: atomic formulas for the cluster $VC_j$
    Output : $\hat{D}_{VC_j}$: set of abstract states for the cluster $VC_j$ with labels
 1  $\hat{D}_{VC_j} \leftarrow \emptyset$;
 2  $N_1 \leftarrow \emptyset$;
 3  $Q \leftarrow \{N_1\}$;
 4  while $Q$ is not empty do
 5      $N_i \leftarrow$ remove element from $Q$;
 6      if $i = k+1$ then
 7          Let $\hat{s}$ be a new abstract state with $\hat{L}(\hat{s}) = N_i$;
 8          $\hat{D}_{VC_j} \leftarrow \hat{D}_{VC_j} \cup \{\hat{s}\}$;
 9      else
10          $N^1_{i+1} \leftarrow N_i \cup \{\varphi_i\}$;
11          if $Inv \wedge \bigwedge_{\varphi \in N^1_{i+1}} \varphi$ is satisfiable then $Q \leftarrow Q \cup \{N^1_{i+1}\}$;
12          $N^2_{i+1} \leftarrow N_i \cup \{\neg\varphi_i\}$;
13          if $Inv \wedge \bigwedge_{\varphi \in N^2_{i+1}} \varphi$ is satisfiable then $Q \leftarrow Q \cup \{N^2_{i+1}\}$;
14      end
15  end
16  return $\hat{D}_{VC_j}$;
```

Example 4.7. *Consider the system T in Listing 4.1 and the variable cluster $VC_1 = \{x, y\}$ with the formula cluster $FC_1 = \{(x < y), (x \doteq y), (y \doteq 2)\}$. The tree can be seen in Figure 4.5. Nodes colored gray are contradicting, so the same five abstract states are obtained as in Example 4.6. The node N^1_3 is already contradicting since $(x < y) \wedge (x \doteq y)$ cannot hold and thus, N^1_4 and N^2_4 are contradicting as well. The node N^7_4 is contradicting because $\neg(x \doteq y) \wedge \neg(x < y)$ implies that $(x > y)$, and $(x > y) \wedge (y \doteq 2)$ implies that $(x > 2)$, but $D_x = \{0, 1, 2\}$.*

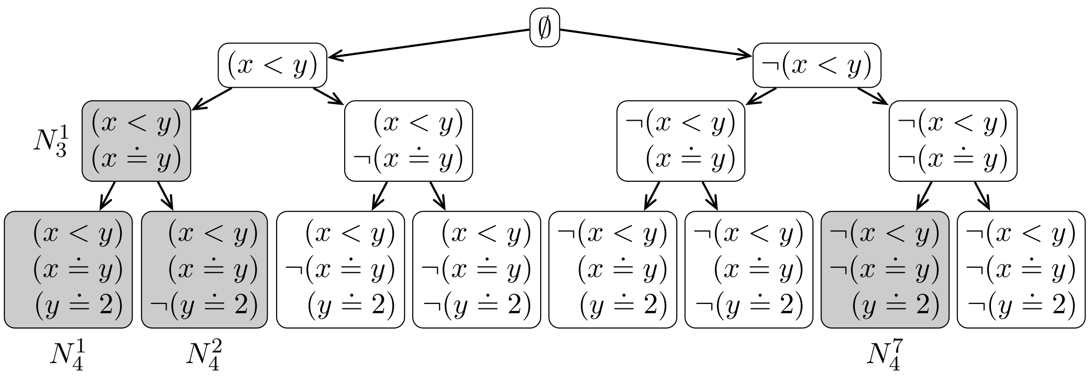


Figure 4.5: *Example on enumerating only the abstract states.*

The abstract states $\hat{S}$ and labeling $\hat{L}$ can thus, be calculated without enumerating concrete states. Since the labels of an abstract state $\hat{s} \in \hat{S}$ are FOL formulas, $s \in \hbar^{-1}(\hat{s})$ can be decided using an SMT solver by checking if $s \models \bigwedge_{l \in \hat{L}(\hat{s})} l$. Therefore, the abstract transition relation $\hat{R}$ and initial states $\hat{I}$ can also be computed without enumerating concrete states with an SMT solver, using queries presented in Section 2.2.2 and the ones below.

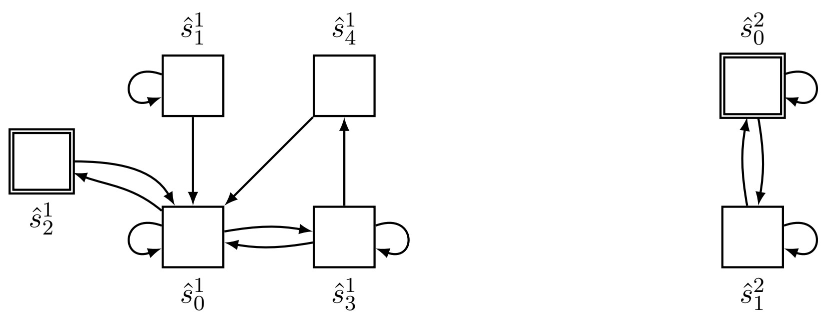
- $(\hat{s}, \hat{s}') \in \hat{R}$ iff $\exists s, s'.\ (s, s') \in R \land s \in \hbar^{-1}(\hat{s}) \land s' \in \hbar^{-1}(\hat{s}')$ is satisfiable.
- $\hat{s} \in \hat{I}$ iff $\exists s.\ s \in I \land s \in \hbar^{-1}(\hat{s})$ is satisfiable.

Clarke et al. represent the abstract Kripke structures symbolically using BDDs [46]. In my approach I represent them explicitly, but for optimization purposes, I do not construct the full abstract Kripke structure $\hat{M} = (\hat{S}, \hat{R}, \hat{L}, \hat{I})$. Instead, I construct a Kripke structure $\hat{M}_i = (\hat{S}_i, \hat{R}_i, \hat{L}_i, \hat{I}_i)$ for each cluster VC_i. Given that $\hbar(s) = (\hat{s}^1, \hat{s}^2, \ldots, \hat{s}^i, \ldots, \hat{s}^m)$, let $\hbar(s)^i$ denote the ith component $\hat{s}^i$. Then $\hat{M}_i$ is defined as follows:

- $\hat{S}_i = \hat{D}_{VC_i}$, i.e., abstract states are the elements of the abstract domain,
- $\hat{s}^i \in \hat{I}_i$ if $\exists s \in I$ with $\hbar(s)^i = \hat{s}^i$, i.e., an initial state exists whose ith component is mapped to $\hat{s}^i$,
- $(\hat{s}^i_1, \hat{s}^i_2) \in \hat{R}_i$ if $\exists (s_1, s_2) \in R$ with $\hbar(s_1)^i = \hat{s}^i_1$ and $\hbar(s_2)^i = \hat{s}^i_2$, i.e., a concrete transition from s_1 to s_2 exists whose ith components are mapped to $\hat{s}^i_1$ and $\hat{s}^i_2$.

The full abstract Kripke structure $\hat{M}$ is constructed on-the-fly by the model checker as it explores the abstract state space.

Example 4.8. *Consider the system in Example 4.6 with $VC_1 = \{x, y\}$ and $VC_2 = \{reset\}$. Recall, that VC_1 and VC_2 had 5 and 2 states respectively. Furthermore, $\hbar$ is a composite abstraction function $(\hbar_1, \hbar_2)$. The abstract Kripke structures can be seen in Figure 4.6. The abstract states are those defined in Example 4.6. The initial state of the original model is $(0, 1, 0)$ and $\hbar(0, 1, 0) = (\hbar_1(0, 1), \hbar_2(0)) = (\hat{s}^1_2, \hat{s}^2_0)$. Therefore $\hat{s}^1_2$ is initial in $\hat{M}_1$ and $\hat{s}^2_0$ is initial in $\hat{M}_2$. The transition from $(1, 2, 1)$ to $(0, 0, 0)$ in the original model is mapped in the following way: $\hbar(1, 2, 1) = (\hbar_1(1, 2), \hbar_2(1)) = (\hat{s}^1_3, \hat{s}^2_1)$ and $\hbar(0, 0, 0) = (\hbar_1(0, 0), \hbar_2(0)) = (\hat{s}^1_0, \hat{s}^2_0)$. Therefore, there is a transition between $\hat{s}^1_3$ and $\hat{s}^1_0$ in $\hat{M}_1$ and between $\hat{s}^2_1$ and $\hat{s}^2_0$ in $\hat{M}_2$. The other transitions are mapped similarly.*



(a) Kripke structure $\hat{M}_1$ for $VC_1 = \{x, y\}$.

(b) Kripke structure $\hat{M}_2$ for $VC_2 = \{reset\}$.

Figure 4.6: *Abstract Kripke structures corresponding to the clusters.*

4.2.2 Model checking

For safety properties of the form $\mathsf{AG}\ \varphi$, model checking amounts to exploring the set of reachable (abstract) states and checking whether every state satisfies φ. Clarke et al. use a symbolic (BDD-based) representation and therefore, symbolic model checking algorithms [46]. In my approach I construct the abstract Kripke structure from its components on-the-fly and use explicit model checking. A sketch of the algorithm can be seen in Algorithm 2. The input of the algorithm is a symbolic transition system T, a list of Kripke structures $\hat{M}_1, \hat{M}_2, \ldots, \hat{M}_m$ corresponding to the clusters and a requirement $\mathsf{AG}\ \varphi$. The output is either an abstract counterexample (in the form of a path) or a positive answer. The set of already explored states is denoted by E, which is initially empty. I loop through each state in the product of the initial states and check whether it is not explored yet and also if it is an initial state in the full Kripke structure $\hat{M}$. This check has to be done since $\hat{I} \subseteq \hat{I}_1 \times \hat{I}_2 \times \ldots \times \hat{I}_m$. For example, consider a system with two variables x, y belonging to different clusters and suppose that the initial states are $(x \doteq 0, y \doteq 1)$ and $(x \doteq 1, y \doteq 0)$. In the cluster of x, states that abstract $x \doteq 0$ and $x \doteq 1$ are initial and similarly for the cluster of y. Thus, a product state abstracting $(x \doteq 0, y \doteq 0)$ is included in $\hat{I}_1 \times \hat{I}_2 \times \ldots \times \hat{I}_m$, but not included in $\hat{I}$.

From each abstract initial state a depth-first search is started using a stack Q. At each iteration the last element $\hat{s}$ of Q is removed and checked whether $\hat{s} \not\models \varphi$. If $\hat{s} \not\models \varphi$ holds, then the actual path $\hat{\pi} = (\hat{s}_0, \ldots, \hat{s})$ is returned as a counterexample. Otherwise all successors of $\hat{s}$ are pushed onto the stack. A check $(\hat{s}, \hat{s}') \in \hat{R}$ is necessary like in the case of initial states, since $\hat{R} \subseteq \hat{R}_1 \times \hat{R}_2 \times \ldots \times \hat{R}_m$. Finally, if no state was reached that violates the requirement, a positive answer is returned. In such cases it can be concluded by Theorem 1 that the concrete Kripke structure M also satisfies $\mathsf{AG}\ \varphi$.

Example 4.9. *Consider the system T in Listing 4.1 with the requirement AG $(x < y \vee reset \doteq 1)$. As mentioned in Example 4.6, there are two clusters with 2 and 5 states. Hence, the abstract state space has $2 \cdot 5 = 10$ states labeled with* $\mathsf{atoms}(T)$ *(or their negations). It can be checked, that an abstract path $\hat{\pi} = (\hat{s}_0, \hat{s}_1, \hat{s}_2)$ exists with the following labels:*

- $\hat{L}(\hat{s}_0) = \{\neg(reset \doteq 1),\ \ (x < y), \neg(x \doteq y), \neg(y \doteq 2)\}$,
- $\hat{L}(\hat{s}_1) = \{\ \ (reset \doteq 1), \neg(x < y),\ \ (x \doteq y), \neg(y \doteq 2)\}$,
- $\hat{L}(\hat{s}_2) = \{\neg(reset \doteq 1), \neg(x < y),\ \ (x \doteq y), \neg(y \doteq 2)\}$.

Consequently, $\hat{\pi}$ is an abstract counterexample since $\hat{s}_0 \in \hat{I}$ and $\hat{s}_2 \not\models (x < y \vee reset \doteq 1)$.

4.2.3 Concretizing the counterexample

The abstract counterexample for safety properties ($\mathsf{AG}\ \varphi$) is a path $\hat{\pi} = (\hat{s}_1, \hat{s}_2 \ldots, \hat{s}_n)$ of abstract states with $\hat{s}_1 \in \hat{I}$ and $\hat{s}_n \not\models \varphi$. The abstraction function $\hbar$ is appropriate,

Algorithm 2: Construct and check abstract state space (clustered CEGAR).

```
Input  : $T$: Symbolic transition system
         $\hat{M}_1, \ldots, \hat{M}_m$: Kripke structures of the clusters
         $\mathsf{AG}\ \varphi$: requirement
Output : "Requirement holds" or an abstract counterexample
1  $E \leftarrow \emptyset$;
2  foreach $\hat{s}_0 \in \hat{I}_1 \times \ldots \times \hat{I}_m$ do
3      if $\neg(\hat{s}_0 \in \hat{I}) \vee \hat{s}_0 \in E$ then continue;
4      $Q \leftarrow \{\hat{s}_0\}$;
5      while $Q \neq \emptyset$ do
6          $\hat{s} \leftarrow \mathrm{pop}(Q)$;
7          if $\hat{s} \notin E$ then
8              $E \leftarrow E \cup \{\hat{s}\}$;
9              if $\hat{s} \not\models \varphi$ then return actual path $(\hat{s}_0, \ldots, \hat{s})$;
10             foreach $\hat{s}'$ with $(\hat{s}, \hat{s}') \in \hat{R}_1 \times \ldots \times \hat{R}_m$ do
11                 if $(\hat{s}, \hat{s}') \in \hat{R}$ then push$(Q, \hat{s}')$;
12             end
13         end
14     end
15 end
16 return "Requirement holds"
```

which means that if an abstract state $\hat{s}_n$ violates φ, then all concrete states in $\hbar^{-1}(\hat{s}_n)$ also violate φ. Therefore, it is clear that if $\hbar^{-1}(\hat{\pi}) \neq \emptyset$ (i.e., $\hat{\pi}$ is a concretizable path) then a concrete counterexample exists. It can be queried from an SMT solver whether $\hbar^{-1}(\hat{\pi}) \neq \emptyset$. However, if $\hbar^{-1}(\hat{\pi}) = \emptyset$, then it is important to know the longest prefix of the abstract counterexample, for which a concrete path exists. The following approach is proposed in [50]. Let φ_k be defined for $1 \leq k \leq n$ over $s_1, s_2, \ldots, s_n$ in the following way.

$$\varphi_k = s_1 \in I \wedge \bigwedge_{1 \leq i \leq k} \hbar(s_i) = \hat{s}_i \wedge \bigwedge_{1 \leq i < k} (s_i, s_{i+1}) \in R$$

It is easy to see that the set of solutions to φ_n equals to $\hbar^{-1}(\hat{\pi})$. Therefore, if φ_n is satisfiable then a concrete counterexample exists. This approach is similar to the *unfolding* technique used in bounded model checking [40], but here, unfolding is restricted by the abstract states. If φ_n is not satisfiable, let $1 \leq f < n$ be the largest index for which φ_f is satisfiable, i.e., the longest prefix of the abstract counterexample that is concretizable. Then $\hat{s}_f$ is called the *failure state*, which provides useful information for the abstraction refinement.

Example 4.10. *Consider the counterexample $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4)$ in Figure 4.7. The concrete transition relation R is indicated by the arrows. Then the solution(s) of*

- φ_1 *are* $\{(s_1), (s_3)\}$,
- φ_2 *are* $\{(s_1, s_4), (s_1, s_5), (s_3, s_6)\}$,

- *φ_3 are $\{(s_1, s_4, s_7), (s_1, s_5, s_7)\}$,*
- *φ_4 is $\emptyset$.*

Hence, $\hat{\pi}$ is spurious and the failure state is $\hat{s}_3$.

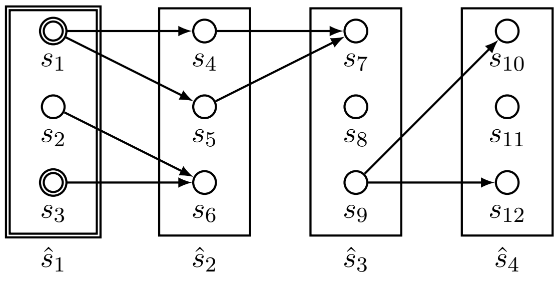


Figure 4.7: *Spurious counterexample.*

4.2.4 Abstraction refinement

When a counterexample $\hat{\pi}$ has no corresponding concrete counterexample, the abstraction has to be refined in order to eliminate the spurious behavior. Example 4.2 and Example 4.3 already presented the rough idea behind abstraction refinement: concrete states (in the failure state) are classified as *dead-end*, *bad* or *irrelevant*. The purpose of refinement is to separate dead-end and bad states.

Definition 4.7 (Dead-end, bad, irrelevant state). Given a spurious counterexample $\hat{\pi} = (\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n)$, the formal definition of dead-end (S_D), bad (S_B) and irrelevant (S_I) states are the following [46, 50]:

$$\begin{aligned} S_D &= \left\{ s_f \,\middle|\, \exists (s_1, s_2, \ldots, s_f) \colon s_1 \in I \wedge \bigwedge_{1 \le i \le f} \hbar(s_i) = \hat{s}_i \wedge \bigwedge_{1 \le i < f} (s_i, s_{i+1}) \in R \right\}, \\ S_B &= \left\{ s \in \hbar^{-1}(\hat{s}_f) \,\middle|\, \exists s' \in \hbar^{-1}(\hat{s}_{f+1}) \colon (s, s') \in R \right\}, \\ S_I &= \hbar^{-1}(\hat{s}_f) \setminus (S_D \cup S_B), \end{aligned}$$

where f is the index of the failure state.

In other words, S_D denotes states in $\hbar^{-1}(\hat{s}_f)$ that are reachable from initial states, S_B denotes states in $\hbar^{-1}(\hat{s}_f)$ that have at least one successor in $\hbar^{-1}(\hat{s}_{f+1})$ and S_I denotes all the other states in $\hbar^{-1}(\hat{s}_f)$. It is obvious that $S_D \cap S_B = \emptyset$, otherwise $\hat{s}_f$ would not be a failure state. The counterexample is spurious due to the transition $(\hat{s}_f, \hat{s}_{f+1}) \in \hat{R}$ introduced by the bad states S_B.

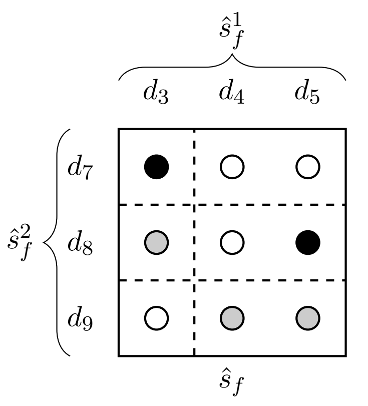
Example 4.11. *Consider the counterexample in Figure 4.7 where the failure state is $\hat{s}_3$. The concrete states are classified as following.*

- *s_7 is a dead-end state since it is reachable from an initial state, but no state in $\hbar^{-1}(\hat{s}_4)$ is a successor of s_7.*

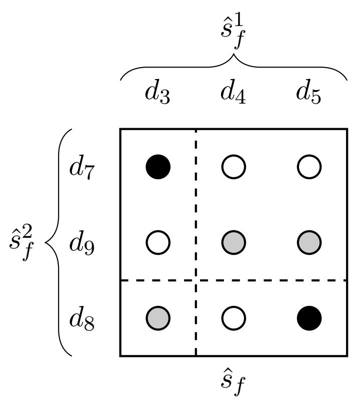
- *s_9 is a bad state since it is not reachable from an initial state, but it has successors (s_{10} and s_{12}) in $\hbar^{-1}(\hat{s}_4)$.*
- *s_8 is irrelevant because it is neither dead-end nor bad.*

Recall that the abstraction function $\hbar = (\hbar_1, \hbar_2, \ldots, \hbar_m)$ is a composite abstraction function, where m is the number of clusters. The failure state $\hat{s}_f = (\hat{s}^1_f, \hat{s}^2_f \ldots, \hat{s}^m_f)$ is a composition of states from each cluster, where each state $\hat{s}^i_f$ is an equivalence class of $\equiv_{\hbar_i}$. Therefore $\hbar$ cannot be refined directly, but only by refining its components $\hbar_i$ [46]. Formally, a composite abstraction function $\equiv'_{\hbar}$ is a refinement of $\equiv_{\hbar}$ if for each component $\equiv'_{\hbar_i} \subseteq \equiv_{\hbar_i}$ holds ($1 \leq i \leq m$), but at least one component exists, where $\equiv'_{\hbar_j} \subset \equiv_{\hbar_j}$ ($1 \leq j \leq m$). This means that some concrete states will no longer be equivalent, yielding new abstract states. The *cost of the refinement* is the number of new abstract states. The goal is to keep the abstraction as coarse as possible, i.e., minimizing the number of new states.

Example 4.12 (from [46]). *Suppose that there are two clusters and the failure state $\hat{s}_f$ corresponds to $(\hat{s}^1_f, \hat{s}^2_f)$ with $\hat{s}^1_f$ abstracting the concrete components d_3, d_4, d_5 (of the first cluster) and $\hat{s}^2_f$ abstracting d_7, d_8, d_9 (from the second cluster). In Figure 4.8 dead-end states are gray, bad states are black and irrelevant states are white. Figure 4.8(a) shows a refinement (indicated by the dashed lines) where $\hat{s}^1_f$ is partitioned into two and $\hat{s}^2_f$ into three new equivalence classes. It can be seen, that dead-end and bad states are successfully separated in the resulting $2 \cdot 3 = 6$ states. However, as Figure 4.8(b) illustrates, if d_3 is already separated from d_4 and d_5, then the separation of d_7 and d_9 is unnecessary. Consequently, a coarser refinement producing only $2 \cdot 2$ new states can also be obtained.*



(a) Refinement with $2 \cdot 3$ new equivalence classes.



(b) Coarser refinement with $2 \cdot 2$ new equivalence classes.

Figure 4.8: *Example on a fine and a coarse refinement.*

Keeping the abstraction as coarse as possible is important to handle the state space explosion problem. However, finding the coarsest refinement is an NP-hard problem [46]. Clarke et al. proposed a heuristic called "PolyRefine", which has polynomial complexity and furthermore, if $S_I = \emptyset$ then it can produce the coarsest refinement [46].

Recall that $S = D_{VC_1} \times D_{VC_2} \times \ldots \times D_{VC_m}$ and a concrete state is defined by an m-tuple $(d_1, d_2, \ldots, d_m)$. Given a set $X \subseteq S$, an index $1 \leq j \leq m$ and a value $a \in D_{VC_j}$ the projection function $\mathsf{proj}(X, j, a)$ [46] is given by

$$\mathsf{proj}(X, j, a) = \{(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \mid (d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in X\}.$$

Informally $\mathsf{proj}(X, j, a)$ is the set of elements in X where the jth component is a but with a removed.

Given a failure state $\hat{s}_f$ and the set of dead-end states S_D, the heuristic "PolyRefine" can be seen in Algorithm 3.

Algorithm 3: PolyRefine.

```
  Input  : $\hat{s}_f$: failure state
           $S_D$: dead-end states
           $(\hbar_1, \hbar_2, \ldots, \hbar_m)$: abstraction functions
  Output : $(\hbar'_1, \hbar'_2, \ldots, \hbar'_m)$: refined abstraction functions
1 for $1 \leq j \leq m$ do
2     $\equiv'_{\hbar_j} \leftarrow \equiv_{\hbar_j}$;
3     for every pair $m_1, m_2 \in \hat{s}^j_f$ do
4         if $\mathsf{proj}(S_D, j, m_1) \neq \mathsf{proj}(S_D, j, m_2)$ then $\equiv'_{\hbar_j} \leftarrow \equiv'_{\hbar_j} \setminus \{(m_1, m_2)\}$;
5     end
6 end
```

The algorithm loops through each component $\hbar_j$ of the abstraction function. Initially the refined equivalence relation $\equiv'_{\hbar_j}$ is the same as $\equiv_{\hbar_j}$. Then, the algorithm checks each pair of concrete states in the jth abstract component $\hat{s}^j_f$ of the failure state. If their projection on the dead-end states is different, the two states can no longer belong to the same equivalence class.

Example 4.13. *Recall the failure state $\hat{s}_f$ in Example 4.12. First $\hat{s}^1_f$ is checked, where* $\mathsf{proj}(S_D, 1, d_3) = \{d_8\}$, $\mathsf{proj}(S_D, 1, d_4) = \{d_9\}$ *and* $\mathsf{proj}(S_D, 1, d_5) = \{d_9\}$. *Thus,* (d_3, d_4) *and* (d_3, d_5) *are removed from* $\equiv_{\hbar_1}$, *i.e.,* d_3 *is separated from* d_4 *and* d_5. *Then* $\hat{s}^2_f$ *is checked, where* $\mathsf{proj}(S_D, 2, d_7) = \emptyset$, $\mathsf{proj}(S_D, 2, d_8) = \{d_3\}$ *and* $\mathsf{proj}(S_D, 2, d_9) = \{d_4, d_5\}$. *Therefore, all three states are separated, yielding the (non-coarsest) refinement of Figure 4.8(a).*

4.3 Visibility-based CEGAR

The visibility-based CEGAR is mainly based on the work of Clarke et al. [50]. The key idea of the approach is to partition variables into a visible and an invisible set. Initial abstraction amounts to determining the visible variables (Section 4.3.1). The abstract model is constructed and checked on-the-fly only taking the visible variables into consideration (Section 4.3.2). An abstract counterexample is concretized in the same way as in the clustered approach (Section 4.3.3). Refinement is achieved by making some of the previously invisible variables visible (Section 4.3.4).

4.3.1 Initial abstraction

Given a symbolic transition system T with a finite set of variables $V = \{v_1, v_2, \ldots, v_n\}$ with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$ and a requirement φ, the concrete Kripke structure $M = (S, L, R, I)$ can be built in the following way. Building S, R and I is presented in Section 2.2.2. Let $\mathsf{atoms}(\varphi)$ be the set of atomic formulas that appear in the requirement φ. Then the labeling $L\colon S \mapsto 2^{\mathsf{atoms}(\varphi)}$ is given by $L(s) = \{\varphi_0 \in \mathsf{atoms}(\varphi) \mid s \models \varphi_0\}$, i.e., states are labeled with the atomic formulas that hold for them.

The set of variables V is partitioned into two sets: *visible* variables, denoted by V_V and *invisible* variables, denoted by V_I. Formally, $V = V_V \cup V_I$ and $V_V \cap V_I = \emptyset$. Intuitively, visible variables are believed to play an important role in verifying the requirement, while invisible variables are not of interest. Without the loss of generality, in the following it is assumed that visible variables are always represented by the first k indices $(1 \le k \le n)$, i.e., $V_V = \{v_1, v_2, \ldots, v_k\}$ and $V_I = \{v_{k+1}, \ldots, v_n\}$. The abstraction function $\hbar\colon S \mapsto \hat{S}$ is defined in the following way, where $(d_1, d_2, \ldots, d_n), (e_1, e_2, \ldots, e_n) \in S$ [50]:

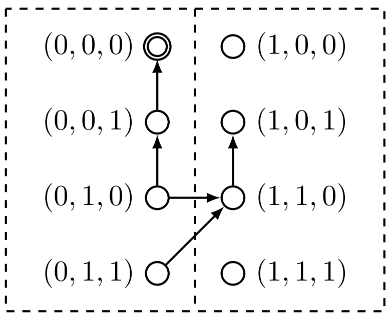
$$(d_1, d_2, \ldots, d_n) \equiv_{\hbar} (e_1, e_2, \ldots, e_n) \text{ iff } \bigwedge_{i=1}^{k} d_i \doteq e_i.$$

Informally, this means that two concrete states are in the same equivalence class (i.e., abstract state) if they have the same value on the visible variables. The set of abstract states is thus $\hat{S} = \{(d_1, d_2, \ldots, d_k) \in D_{v_1} \times D_{v_2} \times \ldots \times D_{v_k} \mid \exists d_{k+1}, d_{k+2}, \ldots, d_n \in D_{v_{k+1}} \times D_{v_{k+2}} \times \ldots \times D_{v_n} : (d_1, d_2, \ldots, d_n) \in S\}$, i.e., assignments to the visible variables from all states.

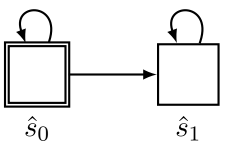
Initially, let $V_V = \bigcup_{\varphi_0 \in \mathsf{atoms}(\varphi)} \mathsf{var}(\varphi_0)$ and $V_I = V \setminus V_V$, i.e., variables appearing in the atomic subformulas of the requirement are visible [50]. This ensures appropriateness of $\hbar$, since states s_1, s_2 with $s_1 \equiv_{\hbar} s_2$ have the same values for variables appearing in the requirement and thus, have the same truth value for each atomic subformula φ_0 of φ, i.e., $s_1 \models \varphi_0 \Leftrightarrow s_2 \models \varphi_0$ holds. This technique is also referred to as *explicit-value analysis* [51].

The abstract states $\hat{S}$ can be calculated by enumerating all the possible evaluations of the visible variables V_V. The abstract labeling $\hat{L}$ is defined as usual, i.e., taking the union of the labels of the concrete states. The abstract transition relation $\hat{R}$ and initial states $\hat{I}$ can be computed using an SMT solver as described for the clustered approach (Section 4.2.1). For optimization purposes however, the abstract Kripke structure is not built explicitly. Instead, it is constructed on-the-fly by the model checker as it explores the abstract state space.

Example 4.14. *Consider a system with variables x, y, z and domains $D_x = D_y = D_z = \{0, 1\}$, having the concrete state space presented in Figure 4.9(a). Let the requirement be $\varphi = \mathsf{AG}\ (x \not\doteq 1)$. The only visible variable is therefore, x. The partitioning of states is presented in Figure 4.9(a), while the abstract state space can be seen in Figure 4.9(b). The abstract state $\hat{s}_0$ corresponds to states with $x \doteq 0$ and $\hat{s}_1$ to states with $x \doteq 1$.*



(a) Concrete state space.



(b) Abstract state space.

Figure 4.9: *Abstraction example based on visible and invisible variables.*

4.3.2 Model checking

As in the clustered approach, model checking of safety properties ($\mathsf{AG}\ \varphi$) amounts to exploring the reachable states and checking whether every state satisfies φ. Clarke et al. use a symbolic model checker [50] for this purpose. In my approach I construct the abstract Kripke structure on-the-fy and use explicit model checking. A sketch of the algorithm can be seen in Algorithm 4. The input of the algorithm is a symbolic transition system T, a set of visible variables V_V and a requirement $\mathsf{AG}\ \varphi$. The output is either an abstract counterexample or a positive answer. The set of already explored states is denoted by E, which is initially empty. I loop through each state in the set of abstract initial states $\hat{I}$. The set $\hat{I}$ can be enumerated on-the-fly by an SMT solver like the set of all abstract states $\hat{S}$, but with an extra assertion that these states also have to satisfy *Init*. From each abstract initial state a depth-first search is started using a stack Q. At each iteration the last element $\hat{s}$ of Q is removed and checked whether $\hat{s} \not\models \varphi$. If $\hat{s} \not\models \varphi$ holds, then the actual path is returned as a counterexample. Otherwise all successors of $\hat{s}$ are pushed onto the stack. Finally, if no state was reached that violates the requirement, a positive answer is returned. In such cases it can be concluded by Theorem 1 that the concrete Kripke structure also satisfies $\mathsf{AG}\ \varphi$.

4.3.3 Concretizing the counterexample

The abstract counterexample can be concretized (unfolded) in the same way as in the clustered approach (Section 4.2.3). If no corresponding concrete counterexample can be found, the failure state provides useful information for abstraction refinement.

Example 4.15. *Consider the system in Example 4.14 and Figure 4.9. The abstract path* $\hat{\pi} = (\hat{s}_0, \hat{s}_1)$ *is a counterexample for the property* $\mathsf{AG}\ (x \neq 1)$. *However, starting from the only concrete initial state* $(0, 0, 0)$ *in* $\hat{s}_0$*, there is no concrete transition to* $\hat{s}_1$. *Therefore* $\hat{s}_0$ *is the failure state,* $(0, 0, 0)$ *is a dead-end state,* $(0, 1, 0)$ *and* $(0, 1, 1)$ *are bad states and* $(0, 0, 1)$ *is irrelevant.*

Algorithm 4: Construct and check abstract state space (visibility-based CEGAR).

```
Input  : $T$: Symbolic transition system
         $V_V$: visible variables
         $\mathsf{AG}\ \varphi$: requirement
Output : “Requirement holds” or an abstract counterexample
1  $E \leftarrow \emptyset$;
2  foreach $\hat{s}_0 \in \hat{I}$ do
3      if $\hat{s}_0 \in E$ then continue;
4      $Q \leftarrow \{\hat{s}_0\}$;
5      while $Q \neq \emptyset$ do
6          $\hat{s} \leftarrow \mathrm{pop}(Q)$;
7          if $\hat{s} \notin E$ then
8              $E \leftarrow E \cup \{\hat{s}\}$;
9              if $\hat{s} \not\models \varphi$ then return actual path $(\hat{s}_0, \dots, \hat{s})$;
10             foreach $\hat{s}'$ with $(\hat{s}, \hat{s}') \in \hat{R}$ do $\mathrm{push}(Q, \hat{s}')$;
11         end
12     end
13 end
14 return “Requirement holds”
```

4.3.4 Abstraction refinement

As in the clustered approach, the purpose of abstraction refinement is to separate the dead-end and bad states. In this case however, separation can be achieved by making a subset $V_I' \subseteq V_I$ of the previously invisible variables visible. Then, refining the abstraction means to move elements of V_I' from V_I to V_V, i.e., $V_I \leftarrow V_V \cup V_I'$ and $V_I \leftarrow V_I \setminus V_I'$. Clarke et al. proposed heuristics based on *integer linear programming* and *decision tree learning* for determining V_I' [50]. In my work, I adopt a different strategy: I generate a formula $\mathscr{I}$ that separates dead-end and bad states using Craig interpolation. Then I simply choose $V_I' = \mathsf{var}(\mathscr{I})$, i.e., I make variables appearing in the interpolant visible.

Dead-end and bad states can be described as in the clustered approach (Definition 4.7):

$$\begin{aligned} S_D &= \left\{ s_f \,\middle|\, \exists \pi = (s_1, s_2, \dots, s_f) \colon s_1 \in I \wedge \bigwedge_{1 \le i \le f} \hbar(s_i) = \hat{s}_i \wedge \bigwedge_{1 \le i < f} (s_i, s_{i+1}) \in R \right\}, \\ S_B &= \left\{ s \in \hbar^{-1}(\hat{s}_f) \,\middle|\, \exists s' \in \hbar^{-1}(\hat{s}_{f+1}) \colon (s, s') \in R \right\}. \end{aligned}$$

For interpolation, let α and β be defined over $s_1, s_2, \dots, s_n$ in the following way.

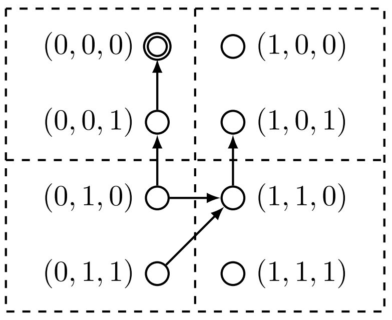
$$\begin{aligned} \alpha &= s_1 \in I \wedge \bigwedge_{1 \le i \le f} \hbar(s_i) = \hat{s}_i \wedge \bigwedge_{1 \le i < f} (s_i, s_{i+1}) \in R \\ \beta &= \hbar(s_{f+1}) = \hat{s}_{f+1} \wedge (s_f, s_{f+1}) \in R \end{aligned}$$

The formula α describes paths leading to dead-end states in $\hbar^{-1}(\hat{s}_f)$, while β describes bad states. Since $\hat{\pi}$ is a spurious counterexample, $\alpha \wedge \beta$ is unsatisfiable. Therefore, a

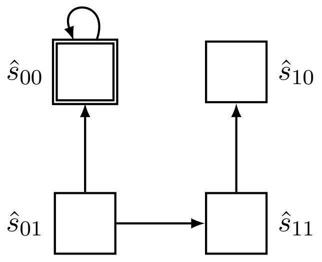
Craig interpolant $\mathscr{I}$ can be calculated. The important properties of the interpolant are the following:

- $\alpha \Rightarrow \mathscr{I}$, i.e., $\mathscr{I}$ describes dead-end states using less information than α,
- $\mathscr{I} \land \beta$ is unsatisfiable, i.e., bad states cannot satisfy $\mathscr{I}$,
- $\mathscr{I}$ uses only common symbols in α and β, which are only variables of s_f.

Example 4.16. *Recall Example 4.15 with Figure 4.9. The failure state is $\hat{s}_0$, thus $\alpha = (s_0 \in I \land \hbar(s_0) = \hat{s}_0)$ and $\beta = (\hbar(s_1) = \hat{s}_1 \land (s_0, s_1) \in R)$. The formula $\mathscr{I} = (y_0 \doteq 0)$ is an interpolant, since $\alpha \Rightarrow \mathscr{I}$, $\mathscr{I} \land \beta$ is unsatisfiable and the variable y_0 corresponds to the common symbol s_0. Therefore, $\mathsf{var}(\mathscr{I}) = \{y\}$, and $V_V = \{x, y\}$ and $V_I = \{z\}$ after the refinement. The refined partitioning and the abstract state space can be seen in Figure 4.10, where the spurious behavior is now eliminated. Note, that $\mathscr{I}' = (y_0 \doteq 0 \land z_0 \doteq 0)$ is also an interpolant, but it would make both y and z visible, yielding a larger abstract state space. The algorithm thus, relies heavily on how "simple" interpolants the solvers can generate.*



(a) Concrete state space.



(b) Abstract state space.

Figure 4.10: *Abstraction refinement by making the variable y visible.*

4.4 Interpolating CEGAR

The interpolating CEGAR algorithm can be regarded as a combination of the previous approaches along with some advanced techniques. It builds the initial abstraction based on explicitly tracked variables and an arbitrary set of initial predicates (Section 4.4.1). Model checking is performed on an explicitly constructed model (Section 4.4.2) with an incremental optimization (Section 4.4.5). An abstract counterexample is concretized similarly to the previous approaches, but with some additional checks (Section 4.4.3). The key of the algorithm is the refinement (Section 4.4.4), which is achieved by extending the set of predicates using Craig or sequence interpolation. Lazy abstraction is also utilized, i.e., only a subset of the abstract states is refined by the new predicates.

4.4.1 Initial abstraction

Given a symbolic transition system T with a finite set of variables $V = \{v_1, v_2, \dots, v_n\}$ with domains $D_{v_1}, D_{v_2}, \dots, D_{v_n}$ and a requirement φ, let $\mathcal{P}$ be a set of arbitrary, quantifier-free FOL predicates over V and let $V_E \subseteq V$ be the set of *explicitly tracked variables*. Without the loss of generality, in the following it is assumed that explicitly tracked variables are represented by the first k indices ($1 \leq k \leq n$), i.e., $V_E = \{v_1, v_2, \dots, v_k\}$. The Kripke structure $M = (S, L, R, I)$ can be constructed as described in Section 2.2.2 with the labeling function $L\colon S \mapsto 2^{\mathcal{P}}$ given by $L(s) = \{\varphi \in \mathcal{P} | s \models \varphi\} \cup \{\neg\varphi | \varphi \in \mathcal{P} \wedge s \not\models \varphi\}$. Note, that states are explicitly labeled with the negations of predicates that they do not satisfy. The motivation behind this is explained later in this section. The abstraction function $\hbar\colon S \mapsto \hat{S}$ is defined in the following way, where $(d_1, d_2, \dots, d_n), (e_1, e_2, \dots, e_n) \in S$:

$$\begin{aligned}(d_1, d_2, \dots, d_n) \equiv_\hbar (e_1, e_2, \dots, e_n) \text{ iff } & \bigwedge_{\varphi \in \mathcal{P}} [(d_1, d_2, \dots, d_n) \models \varphi \Leftrightarrow (e_1, e_2, \dots, e_n) \models \varphi] \\ \wedge & \bigwedge_{i=1}^{k} d_i \doteq e_i.\end{aligned}$$

Informally, this means that two states are mapped to the same abstract state if they cannot be distinguished by the predicates in $\mathcal{P}$ and they have the same values for the explicitly tracked variables. The abstract labeling function was defined as $\hat{L}(\hat{s}) = \bigcup_{s \in \hbar^{-1}(\hat{s})} L(s)$ in the generic framework (Section 4.1), i.e., the union of the labels of concrete states. In the interpolating CEGAR approach however, abstract states also carry information about the values of explicitly tracked variables V_E. Therefore, $\hat{L}$ also associates abstract states with labels of the form $(v_i \doteq d_i)$ for each $v_i \in V_E$, where $d_i \in D_{v_i}$ is the value of v_i in the abstract state. For example, if $V_E = \{x\}$ and $D_x = \{0, 1, 2\}$, abstract states are also labeled with $x \doteq 0$, $x \doteq 1$ or $x \doteq 2$ besides the predicates.

The rationale behind combining predicates and explicitly tracked variables is that both approaches have different advantages and disadvantages. For example, a variable with an infinite domain cannot be tracked explicitly. On the other hand, a variable with a small domain (e.g., the program counter) may yield many predicates in many steps. In such cases it is more efficient to keep track of the variable explicitly.

The initial abstract Kripke structure $\hat{M} = (\hat{S}, \hat{R}, \hat{L}, \hat{I})$ can be built without explicitly calculating M in the following way. If explicitly tracked variables are ignored, the abstract states and labeling can be calculated using predicate abstraction in the same way as one cluster in the clustered CEGAR approach (Section 4.2.1). Therefore, let $\hat{S}_0$ and $\hat{L}_0$ denote the set of abstract states and labels produced by Algorithm 1 with the set of predicates $\mathcal{P}$. Then Algorithm 5 creates $\hat{S}$ and $\hat{L}$ by extending $\hat{S}_0$ and $\hat{L}_0$ with the explicitly tracked variables. Algorithm 5 loops through the abstract states in $\hat{S}_0$ and for each state $\hat{s}_0$ it loops through all possible evaluations of the explicitly tracked variables. A new state $\hat{s}$ is created by appending the concrete values to $\hat{s}_0$ and the labels are also updated. If the labels are not contradicting, the state is added to $\hat{S}$.

Algorithm 5: Construct initial abstract states and labeling (interpolating CEGAR).

```
Input  : $\hat{S}_0$: abstract states from predicates $\mathcal{P}$
         $\hat{L}_0$: abstract labeling over predicates $\mathcal{P}$
         $V_E$: explicitly tracked variables
Output : $\hat{S}$: abstract states and labeling with explicitly tracked variables
1  $\hat{S} \leftarrow \emptyset$;
2  $\hat{L} \leftarrow \hat{L}_0$;
3  foreach $\hat{s}_0 \in \hat{S}_0$ do
4      foreach $(d_1, d_2, \ldots, d_k) \in D_{v_1} \times D_{v_2} \times \ldots \times D_{v_k}$ do
5          $\hat{s} \leftarrow (\hat{s}_0, d_1, \ldots, d_k)$;
6          $\hat{L}(\hat{s}) \leftarrow \hat{L}_0(\hat{s}) \cup \{(v_1 \doteq d_1)\} \cup \{(v_2 \doteq d_2)\} \cup \ldots \cup \{(v_k \doteq d_k)\}$;
7          if $Inv \wedge \bigwedge_{l \in \hat{L}(\hat{s})} l$ is satisfiable then $\hat{S} \leftarrow \hat{S} \cup \{\hat{s}\}$;
8      end
9  end
10 return $\hat{S}$
```

Example 4.17. *Suppose, that variables are $V = \{x, y\}$ with domains $D_x = D_y = \{0, 1\}$, the only predicate is $\mathcal{P} = \{(x < y)\}$ and the only explicitly tracked variable is $V_E = \{x\}$. Considering only the predicates, there are two abstract states $\hat{s}, \hat{t} \in \hat{S}_0$ with $\hat{L}_0(\hat{s}) = \{(x < y)\}$ and $\hat{L}_0(\hat{t}) = \{\neg(x < y)\}$. By considering the concrete values of x there are thus, $2 \cdot |D_x| = 4$ abstract states:*

- $\hat{s}_0$ *with* $\hat{L}(\hat{s}_0) = \{\ \ (x < y), (x \doteq 0)\}$,
- $\hat{s}_1$ *with* $\hat{L}(\hat{s}_1) = \{\ \ (x < y), (x \doteq 1)\}$,
- $\hat{t}_0$ *with* $\hat{L}(\hat{t}_0) = \{\neg(x < y), (x \doteq 0)\}$,
- $\hat{t}_1$ *with* $\hat{L}(\hat{t}_1) = \{\neg(x < y), (x \doteq 1)\}$.

However, labels of $\hat{s}_1$ cannot be satisfied, since $(x < y) \wedge (x \doteq 1) \Rightarrow (1 < y)$, but $D_y = \{0, 1\}$.

The abstract transition relation $\hat{R}$ and initial states $\hat{I}$ can be calculated with an SMT solver similarly to the clustered algorithm. In this approach however, I construct the full abstract Kripke structure $\hat{M}$ since I expect only a few predicates and explicitly tracked variables with small domains. This is motivated by the fact that usually in software model checking, the only explicitly tracked variable is the program counter and the initial set of predicates is empty [48].

Labeling revisited

Given an ACTL* formula φ to be checked, if the set of predicates $\mathcal{P}$ contains the atomic subformulas of φ (i.e., $\mathsf{atoms}(\varphi) \subseteq \mathcal{P}$) then the abstraction function $\hbar$ defined above is appropriate. However, since predicates are arbitrary, it is also possible that $\mathsf{atoms}(\varphi) \not\subseteq \mathcal{P}$, i.e., the model and the requirement is defined over different atomic propositions.[3] The rest of this section discusses this case.

It is rarely meaningful to define the model and the requirement over different sets of atomic propositions if they are uninterpreted symbols. For example, $\mathsf{AG}\ (\text{cat} \vee \text{dog})$ would always evaluate to false over the model of a traffic light because no state is labeled with “cat” or “dog”. In my case however, labels are FOL predicates so it is possible to reason about a model that has different labels than the requirement. For example, if a state $\hat{s}$ is labeled with $(x > 5)$, it would be intuitive to say that $\hat{s} \models (x > 0)$ even though $\hat{s}$ is not explicitly labeled with $(x > 0)$.

Formally, let T be a symbolic transition system with a set of variables V, let $\mathcal{P}$ be the set of predicates over V that the abstract Kripke structure $\hat{M}$ is labeled with, let φ_0 be a FOL literal over V (i.e. a predicate or its negation) and let ψ_1, ψ_2 be FOL formulas (without temporal operators). The relation $\models$ is then defined in the following way for safety properties on abstract Kripke structures.

1. $(\hat{M}, \hat{s}) \models \varphi_0$ iff $\bigwedge_{l \in \hat{L}(\hat{s})} l \Rightarrow \varphi_0$.
2. $(\hat{M}, \hat{s}) \models \psi_1 \wedge \psi_2$ iff $(\hat{M}, \hat{s}) \models \psi_1 \wedge (\hat{M}, \hat{s}) \models \psi_2$.
3. $(\hat{M}, \hat{s}) \models \psi_1 \vee \psi_2$ iff $(\hat{M}, \hat{s}) \models \psi_1 \vee (\hat{M}, \hat{s}) \models \psi_2$.
4. $(\hat{M}, \hat{s}) \models \psi_1 \rightarrow \psi_2$ iff $(\hat{M}, \hat{s}) \models \psi_1 \rightarrow (\hat{M}, \hat{s}) \models \psi_2$.
5. $(\hat{M}, \hat{s}) \models \psi_1 \leftrightarrow \psi_2$ iff $[(\hat{M}, \hat{s}) \models \psi_1 \wedge (\hat{M}, \hat{s}) \models \psi_2] \vee [(\hat{M}, \hat{s}) \models \neg\psi_1 \wedge (\hat{M}, \hat{s}) \models \neg\psi_2]$.
6. $(\hat{M}, \hat{s}) \models \mathsf{AG}\ \psi_1$ iff $(\hat{M}, \hat{s}_i) \models \psi_1$ holds for each state $\hat{s}_i \in \hat{\pi}$ of each path $\hat{\pi} = (\hat{s}_0, \hat{s}_1, \ldots)$ with $\hat{s}_0 = \hat{s}$.

The definition above is similar to the classical definition of $\models$ (Section 2.3.5). However, there are some differences.

- A state $\hat{s}$ satisfies a literal φ_0 if the conjunction of its labels implies φ_0.
- There is no corresponding rule for negation, i.e., $\hat{s} \not\models \varphi_0$ does not imply that $\hat{s} \models \neg\varphi_0$. If a FOL formula ψ contains negations, $\models$ can be decided by transforming ψ into NNF, where negations only apply to atomic predicates and the first rule for literals can be used.

Lemma 1. Let φ be a FOL formula and let the abstraction $\hbar$ be defined as above. If $\hat{s} \models \varphi$ holds for an abstract state $\hat{s}$ then $s \models \varphi$ also holds for all concrete states $s \in \hbar^{-1}(\hat{s})$.

Proof. *Suppose that $\hat{s} \models \varphi$, but a state $s \in \hbar^{-1}(\hat{s})$ exists with $s \not\models \varphi$. Since concrete states are evaluations of the variables, if $s \not\models \varphi$, then $s \models \neg\varphi$ must hold. However, $s \models l$ for each label $l \in \hat{L}(\hat{s})$ since concrete states share the label of the abstract state. Therefore, $s \models \neg\varphi \wedge \bigwedge_{l \in \hat{L}(\hat{s})} l$, i.e., $\neg\varphi \wedge \bigwedge_{l \in \hat{L}(\hat{s})} l$ is satisfiable. Using De Morgan's rules,*

[3]As an extreme case, it is also possible that $\mathcal{P} = \{\top\}$, i.e., the whole concrete state space is mapped to a single abstract state with the label $\top$.

this can be rewritten as $\neg(\neg(\bigwedge_{l\in\hat{L}(\hat{s})} l) \wedge \varphi)$*, which is* $\neg((\bigwedge_{l\in\hat{L}(\hat{s})} l) \rightarrow \varphi)$*. This expression is satisfiable, thus by the definition of implication,* $(\bigwedge_{l\in\hat{L}(\hat{s})} l) \not\Rightarrow \varphi$*, i.e.,* $\hat{s} \not\models \varphi$*.*

The opposite direction of Lemma 1 is that if at least one state $s \in \hbar^{-1}(\hat{s})$ exists with $s \not\models \varphi$ (i.e., $s \models \neg\varphi$), then $\hat{s} \not\models \varphi$.

Example 4.18. *Consider a state* $\hat{s}$ *with* $\hat{L}(\hat{s}) = \{x > 5\}$*. Then* $\hat{s} \models (x > 0)$ *but* $\hat{s} \not\models (x \doteq 10)$ *and also* $\hat{s} \not\models \neg(x \doteq 10)$*. Moreover,* $\hat{s}$ *can abstract the concrete states* $s_1 = (x \doteq 10)$ *and* $s_2 = (x \doteq 9)$ *where* $s_1 \models (x \doteq 10)$ *and* $s_2 \models \neg(x \doteq 10)$*.*

The previous example also emphasizes why it is important to explicitly label a concrete state s with a literal $\neg\varphi$ if $s \not\models \varphi$. The following theorem states that the abstraction is an over-approximation, i.e., no false positives are possible.

Theorem 2. Let $\mathcal{P}$ be an arbitrary set of FOL predicates over V, let φ be a FOL formula over V and let $\hat{M}$ be the abstract Kripke structure generated from the concrete Kripke structure M by the abstraction function $\hbar$ defined in this section. Then $\hat{M} \models \mathsf{AG}\ \varphi \Rightarrow M \models \mathsf{AG}\ \varphi$.

Proof. *If* $\hat{M} \models \mathsf{AG}\ \varphi$ *then* $\hat{s} \models \varphi$ *for all reachable abstract states* $\hat{s} \in \hat{S}$*. If a concrete state* $s \in S$ *is reachable with some path* $\pi = (s_0, s_1, \ldots, s)$*, then the abstract state* $\hbar(s)$ *is also reachable with the path* $\hat{\pi} = (\hbar(s_0), \hbar(s_1), \ldots, \hbar(s))$*. Therefore,* $\hbar(s) \models \varphi$ *also holds. Since* $\hbar(s) \models \varphi$*, it can be concluded that* $s \models \varphi$ *by using Lemma 1. This means that* φ *holds for all reachable concrete states, i.e.,* $M \models \mathsf{AG}\ \varphi$*.*

4.4.2 Model checking

Since the requirement is restricted to safety properties ($\mathsf{AG}\ \varphi$) and the abstract Kripke structure is already constructed, model checking amounts to traversing the abstract states $\hat{S}$ and checking if a state $\hat{s}$ with $\hat{s} \not\models \varphi$ is reachable. This is done using depth-first search, which either returns a positive answer ($\hat{M} \models \mathsf{AG}\ \varphi$) or an abstract counterexample. In the former case it is concluded by Theorem 2 that $M \models \mathsf{AG}\ \varphi$.

I also propose an optimization, namely iterative model checking, i.e., a subset of the explored abstract state space does not have to be traversed again after refining the abstraction. This optimization is presented in Section 4.4.5 after introducing abstraction refinement.

4.4.3 Concretizing the counterexample

The abstract counterexample $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n)$ for the formula $\mathsf{AG}\ \varphi$ can be concretized similarly to the clustered CEGAR approach. However, due to the arbitrary set of predicates, $\hat{s}_n \not\models \varphi$ only means that at least one concrete state $s_n \in \hbar^{-1}(\hat{s}_n)$ exists, for which $s_n \not\models \varphi$ holds. However, there can also be states in $\hbar^{-1}(\hat{s}_n)$, where φ holds. Consequently, it has to be checked with an extra condition whether the last concrete state $s_n \in \hbar^{-1}(\hat{s}_n)$ of the concrete path is such that $s_n \not\models \varphi$.

As in the clustered approach, I use an SMT solver to check whether the counterexample $\hat{\pi}$ is concretizable. Let φ_k be defined for $1 \leq k \leq n$ over $s_1, s_2, \ldots, s_n$ in the same way, i.e.,

$$\varphi_k = s_1 \in I \wedge \bigwedge_{1 \leq i \leq k} \hbar(s_i) = \hat{s}_i \wedge \bigwedge_{1 \leq i < k} (s_i, s_{i+1}) \in R$$

If φ_n is not satisfiable then the largest index f with φ_f being satisfiable is the index of the failure state. On the other hand, if φ_n is satisfiable it is still possible that a concrete path $(s_1, s_2, \ldots, s_n)$ is obtained where $s_n \models \varphi$. Thus, an additional formula $\varphi_{n+1} = s_n \not\models \varphi$ has to be checked. If $\varphi_n \wedge \varphi_{n+1}$ is satisfiable then $(s_1, s_2, \ldots, s_n)$ is a concrete counterexample, otherwise the failure state is $\hat{s}_n$.

Example 4.19. *Consider a symbolic transition system T with $V = \{x, y\}$, $D_x = \{0, 1, 2, 3\}$ and $D_y = \{0, 1\}$. The concrete Kripke structure M can be seen in Figure 4.11(a). Suppose, that $\mathcal{P} = \{x < 2, y \doteq 1\}$ and the requirement is AG $(x \neq 3 \vee y \neq 0)$, i.e., only the state $(3, 0)$ violates the requirement. There are thus $2^{|\mathcal{P}|} = 4$ abstract states. Partitioning by $\mathcal{P}$ is indicated by dashed lines in Figure 4.11(a). The corresponding abstract Kripke structure $\hat{M}$ can be seen in Figure 4.11(b) with the following labeling.*

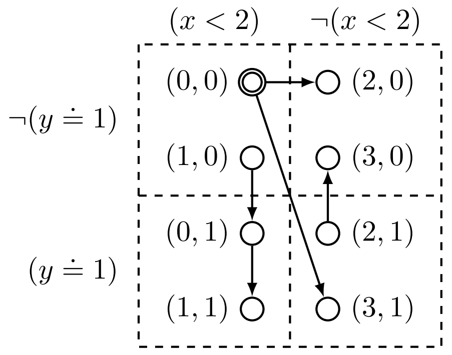
- $\hat{L}(\hat{s}_0) = \{\ \ (x < 2), \neg(y \doteq 1)\}$
- $\hat{L}(\hat{s}_1) = \{\ \ (x < 2), \ \ (y \doteq 1)\}$
- $\hat{L}(\hat{s}_2) = \{\neg(x < 2), \neg(y \doteq 1)\}$
- $\hat{L}(\hat{s}_3) = \{\neg(x < 2), \ \ (y \doteq 1)\}$

Only the abstract state $\hat{s}_2$ violates the requirement (i.e., $\hat{s}_2 \not\models (x \neq 3 \vee y \neq 0)$), which is reachable by the paths $\hat{\pi}_1 = (\hat{s}_0, \hat{s}_2)$ and $\hat{\pi}_2 = (\hat{s}_0, \hat{s}_3, \hat{s}_2)$. Consider first $\hat{\pi}_1$: starting from the initial state $(0, 0)$ only $(2, 0)$ can be reached in $\hbar^{-1}(\hat{s}_2)$, for which $(x \neq 3 \vee y \neq 0)$ holds. Therefore, $\hat{s}_2$ is the failure state, $(2, 0)$ is a dead-end state and $(3, 0)$ is a bad state, causing the spurious counterexample.

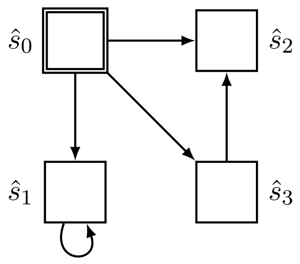
Consider now $\hat{\pi}_2$: starting from $(0, 0)$ only $(3, 1)$ can be reached in $\hbar^{-1}(\hat{s}_3)$, but $(3, 1)$ has no transition to $\hat{s}_2$. Therefore, $\hat{s}_3$ is the failure state, $(3, 1)$ is a dead-end state and $(2, 1)$ is a bad state.

4.4.4 Abstraction refinement

When an abstract counterexample $\hat{\pi}$ has no corresponding concrete counterexample, the abstraction has to be refined. As in the clustered approach, the goal is to separate dead-end and bad states into different abstract states.



(a) Concrete state space.



(b) Abstract state space.

Figure 4.11: *Abstraction example in the interpolating CEGAR.*

Definition 4.8 (Dead-end, bad, irrelevant state). Given a spurious counterexample $\hat{\pi} = (\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n)$, the formal definition of dead-end (S_D), bad (S_B) and irrelevant (S_I) states are the following:

$$
\begin{aligned}
S_D &= \left\{ s_f \,\middle|\, \exists \pi = (s_1, s_2, \ldots, s_f)\colon s_1 \in I \wedge \bigwedge_{1 \le i \le f} \hbar(s_i) = \hat{s}_i \wedge \bigwedge_{1 \le i < f} (s_i, s_{i+1}) \in R \right\}, \\
S_B &= \begin{cases} \{s \in \hbar^{-1}(\hat{s}_f) \mid \exists s' \in \hbar^{-1}(\hat{s}_{f+1})\colon (s, s') \in R\} & \text{if } f < n, \\ \{s \in \hbar^{-1}(\hat{s}_n) \mid s \not\models \varphi\} & \text{if } f = n, \end{cases} \\
S_I &= \hbar^{-1}(\hat{s}_f) \setminus \{S_D \cup S_B\},
\end{aligned}
$$

where f is the index of the failure state and $\mathsf{AG}\ \varphi$ is the requirement.

The definition of S_D and S_I is the same as in the clustered approach. In contrast, S_B has a different definition if the failure state is the last one, i.e., $f = n$. In such cases S_B consists of the states that violate φ. It is easy to see that $S_D \cap S_B = \emptyset$ in this case as well, otherwise $\hat{s}_f$ would not be a failure state.

As its name suggests, the interpolating CEGAR algorithm uses interpolants to refine the abstraction. The algorithm can be configured to use either Craig interpolants to refine only the failure state or interpolation sequences to refine all states of the counterexample.

Refinement with Craig interpolation

Let α and β be defined in the following way over $s_1, s_2, \ldots, s_n$.

$$
\begin{aligned}
\alpha &= s_1 \in I \wedge \bigwedge_{1 \le i \le f} \hbar(s_i) = \hat{s}_i \wedge \bigwedge_{1 \le i < f} (s_i, s_{i+1}) \in R \\
\beta &= \begin{cases} \hbar(s_{f+1}) = \hat{s}_{f+1} \wedge (s_f, s_{f+1}) \in R & \text{if } f < n \\ s_n \not\models \varphi & \text{if } f = n \end{cases}
\end{aligned}
$$

The formula α describes paths leading to dead-end states in $\hbar^{-1}(\hat{s}_f)$ as in the visibility-based CEGAR, while β describes bad states. If $f < n$, bad states are those having a transition to the next state, otherwise bad states are those violating the requirement. Since $\hat{\pi}$ is a spurious counterexample, $\alpha \wedge \beta$ is unsatisfiable. Hence, a Craig interpolant $\mathscr{I}$ can be calculated. The important properties of the interpolant are the following.

- $\alpha \Rightarrow \mathscr{I}$, i.e., $\mathscr{I}$ describes the dead-end states but using less information than α.
- $\mathscr{I} \wedge \beta$ is unsatisfiable, i.e., bad states cannot satisfy $\mathscr{I}$.
- $\mathscr{I}$ uses only common symbols in α and β. If $f < n$ only variables of s_f are the common symbols. Otherwise (if $f = n$) only variables of s_n are the common symbols. Thus, in both cases $\mathscr{I}$ corresponds only to the failure state.

There are two possibilities to refine the abstraction using $\mathscr{I}$.

1. A new abstraction function $\hbar'$ can be calculated in the same way as the initial abstraction but now on the predicate set $\mathcal{P} \cup \{\mathscr{I}\}$. This means that the number of possible states is doubled. Each previous state can correspond to two possible new states by appending the label $\mathscr{I}$ and $\neg\mathscr{I}$. The failure state $\hat{s}_f$ is also split, so the spurious counterexample is eliminated.

2. $\hbar'$ can also be calculated by only replacing $\hat{s}_f$ with $\hat{s}_{f1}$ and $\hat{s}_{f2}$ obtained by adding $\mathscr{I}$ and $\neg\mathscr{I}$ to the labels of $\hat{s}_f$.

In my work I only adopt the latter option to keep the abstract state space as coarse as possible. This approach is called *lazy abstraction* [51,55].

Example 4.20. *Consider the system in Example 4.19 and Figure 4.11 with the requirement* $\mathsf{AG}\ (x \neq 3 \vee y \neq 0)$ *and the counterexample* $\hat{\pi}_1 = (\hat{s}_0, \hat{s}_2)$*. The failure state is* $\hat{s}_2$*, thus interpolation is defined over* s_0, s_2 *in the following way:*

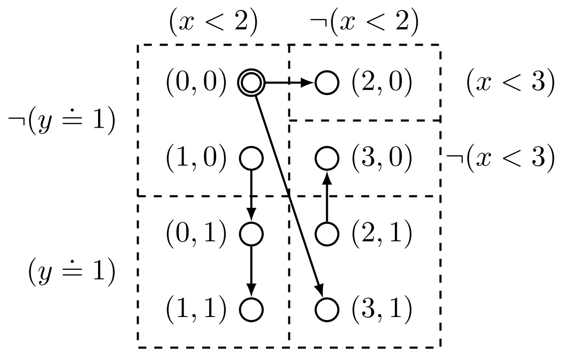
- $\alpha = s_0 \in I \wedge \hbar(s_0) = \hat{s}_0 \wedge \hbar(s_2) = \hat{s}_2 \wedge (s_0, s_2) \in R$,
- $\beta = s_2 \not\models (x_2 \neq 3 \vee y_2 \neq 0)$.

Since s_2 *is a concrete state,* β *can be rewritten as* $\beta = s_2 \models (x_2 \doteq 3 \wedge y_2 \doteq 0)$ *It can be seen that the formula* $\mathscr{I} = (x_2 < 3)$ *is an interpolant, since* $\alpha \Rightarrow \mathscr{I}$*,* $\beta \wedge \mathscr{I}$ *is unsatisfiable and* $\mathscr{I}$ *corresponds to the common variables, namely variables of* s_2*. The new partitioning and the refined abstract state space can be seen in Figure 4.12.*

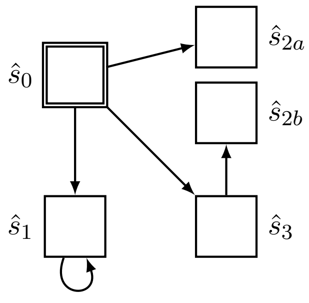
However, the path $\hat{\pi}_2 = (\hat{s}_0, \hat{s}_3, \hat{s}_{2b})$ *is still a counterexample, where the failure state is* $\hat{s}_3$*. Interpolation is therefore, defined over* s_0, s_3, s_{2b} *in the following way:*

- $\alpha = s_0 \in I \wedge \hbar(s_0) = \hat{s}_0 \wedge \hbar(s_3) = \hat{s}_3 \wedge (s_0, s_3) \in R$,
- $\beta = \hbar(s_{2b}) = \hat{s}_{2b} \wedge (s_3, s_{2b}) \in R$.

> *It can be seen that the formula $\mathscr{I}' = (x_3 \doteq 3)$ is an interpolant, corresponding to $\hat{s}_3$. The new partitioning and the refined abstract state space can be seen in Figure 4.13.*

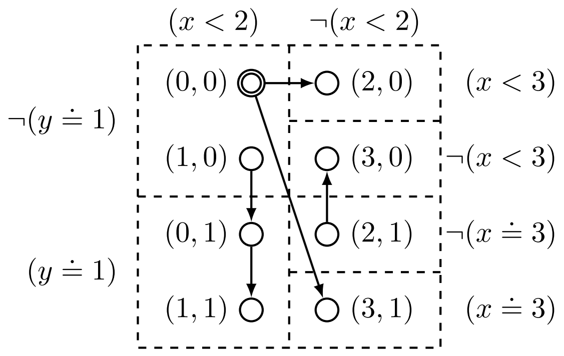


(a) Concrete state space.

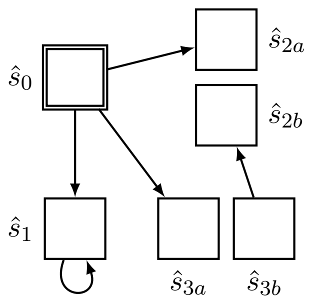


(b) Abstract state space.

Figure 4.12: *Refinement example with Craig interpolant.*



(a) Concrete state space.



(b) Abstract state space.

Figure 4.13: *Refinement example with Craig interpolant.*

Refinement with interpolation sequences

Interpolation sequences do not require the failure state $\hat{s}_f$ to be determined, since they are calculated for the whole counterexample. Let $\alpha_1, \alpha_2, \ldots, \alpha_{n+1}$ be defined in the following way over $s_1, s_2, \ldots, s_n$.

$$\alpha_i = \begin{cases} s_1 \in I \wedge \hbar(s_1) = \hat{s}_1 & \text{if } i = 1 \\ \hbar(s_i) = \hat{s}_i \wedge (s_{i-1}, s_i) \in R & \text{if } 1 < i \leq n \\ s_n \not\models \varphi & \text{if } i = n+1 \end{cases}$$

The formula α_1 describes initial states in $\hbar^{-1}(\hat{s}_1)$, while $\alpha_2, \alpha_3, \ldots, \alpha_n$ describe states in $\hbar^{-1}(\hat{s}_2)$, $\hbar^{-1}(\hat{s}_3), \ldots, \hbar^{-1}(\hat{s}_n)$ that are reachable from the initial states. α_{n+1} describes states violating the requirement. Since $\hat{\pi}$ is a spurious counterexample, $\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_{n+1}$ is unsatisfiable. Thus, an interpolation sequence $\mathscr{I}_0, \mathscr{I}_1, \ldots \mathscr{I}_{n+1}$ exists with the following properties:

- $\mathscr{I}_0 = \top$, $\mathscr{I}_{n+1} = \bot$, i.e., interpolants that do not correspond to any state carry no information,

- $\mathscr{I}_j \land \alpha_{j+1} \Rightarrow \mathscr{I}_{j+1}$ for $0 \leq j \leq n$, i.e., the interpolants together do not allow bad states,
- $\mathscr{I}_j$ refers only to the common symbols of $\alpha_1, \ldots, \alpha_j$ and $\alpha_{j+1}, \ldots, \alpha_{n+1}$, i.e., the variables of s_j.

The refined abstraction $\hbar'$ in case of sequence interpolation is calculated by replacing each $\hat{s}_i$ $(1 \leq i \leq n)$ with $\hat{s}_{i1}$ and $\hat{s}_{i2}$ obtained by adding $\mathscr{I}_i$ and $\neg\mathscr{I}_i$ to the labels of $\hat{s}_i$ respectively. It may occur that $\mathscr{I}_i = \top$ or $\mathscr{I}_i = \bot$ for some $1 \leq i \leq n$. In this case the corresponding abstract state $\hat{s}_i$ is not split.

Example 4.21. *Consider a symbolic transition system with variables $V = \{x, y, z\}$. Suppose, that the requirement is AG $(x \neq 5)$ and the abstract counterexample in Figure 4.14(a) is produced by the model checker. It can be seen that this counterexample is spurious, since $(5, 0, 0)$ cannot be reached from $(0, 0, 0)$. Therefore, $\alpha_1, \alpha_2, \ldots, \alpha_5$ is defined over s_1, s_2, s_3, s_4 for sequence interpolation in the following way:*

- $\alpha_1 = s_1 \in I \land \hbar(s_1) = \hat{s}_1$,
- $\alpha_2 = \hbar(s_2) = \hat{s}_2 \land (s_1, s_2) \in R$,
- $\alpha_3 = \hbar(s_3) = \hat{s}_3 \land (s_2, s_3) \in R$,
- $\alpha_4 = \hbar(s_4) = \hat{s}_4 \land (s_3, s_4) \in R$,
- $\alpha_5 = s_4 \not\models (x_4 \neq 5)$ *(which can be rewritten as $s_4 \models (x_4 \doteq 5)$).*

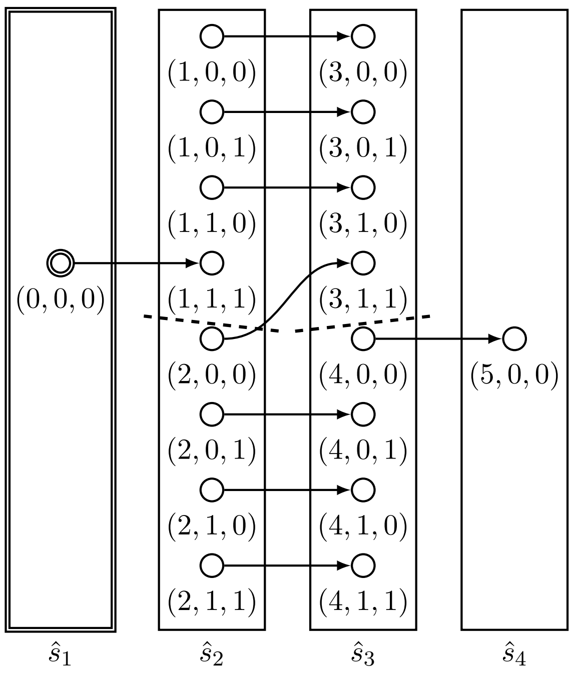
The sequence $\mathscr{I}_0 = \top$, $\mathscr{I}_1 = \top$, $\mathscr{I}_2 = (x_2 < 2)$, $\mathscr{I}_3 = (x_3 < 3)$, $\mathscr{I}_4 = \bot$, $\mathscr{I}_5 = \bot$ is an interpolation sequence, since:

- $\mathscr{I}_0 = \top, \mathscr{I}_5 = \bot$,
- $\top \land s_1 \in I \land \hbar(s_1) = \hat{s}_1 \Rightarrow \top$ $(\mathscr{I}_0 \land \alpha_1 \Rightarrow \mathscr{I}_1)$,
- $\top \land \hbar(s_2) = \hat{s}_2 \land (s_1, s_2) \in R \Rightarrow x_2 < 2$ $(\mathscr{I}_1 \land \alpha_2 \Rightarrow \mathscr{I}_2)$,
- $x_2 < 2 \land \hbar(s_3) = \hat{s}_3 \land (s_2, s_3) \in R \Rightarrow x_3 < 3$ $(\mathscr{I}_2 \land \alpha_3 \Rightarrow \mathscr{I}_3)$,
- $x_3 < 3 \land \hbar(s_4) = \hat{s}_4 \land (s_3, s_4) \in R \Rightarrow \bot$ $(\mathscr{I}_3 \land \alpha_4 \Rightarrow \mathscr{I}_4)$,
- $\bot \land s_4 \not\models (x_4 \neq 5) \Rightarrow \bot$ $(\mathscr{I}_4 \land \alpha_5 \Rightarrow \mathscr{I}_5)$,
- *each $\mathscr{I}_i$ corresponds to the proper common variables.*

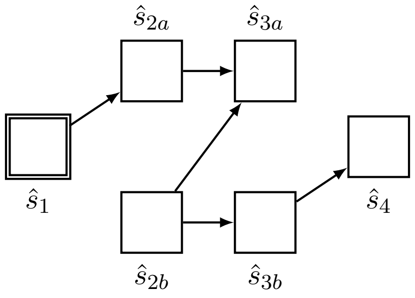
Therefore, $\hat{s}_1$ and $\hat{s}_4$ are not split, $\hat{s}_2$ is split with the predicate $(x < 2)$ and $\hat{s}_3$ with $(x < 3)$ as the dashed lines indicate. The abstract states after the refinement can be seen in Figure 4.14(b). It is clear that the spurious counterexample is eliminated. It can also be seen that both refinements are required.

Suppose now, that Craig interpolation is applied for the same problem. The failure state is $\hat{s}_2$, where $(1, 1, 1)$ is a dead-end state, while other states are bad. Therefore, $(1, 1, 1)$ has to be separated from all other states with an interpolant. However, this requires all three variables (e.g., $\mathscr{I} = (x_2 \doteq 1 \land y_2 \doteq 1 \land z_2 \doteq 1)$), since $(1, 1, 1)$ is not distinguishable with

> *only two variables. In contrast, sequence interpolation could be solved with two predicates containing only x.*



(a) Abstract counterexample.



(b) Abstraction refinement.

Figure 4.14: *Refinement example with sequence interpolant.*

After applying Craig or sequence interpolation, the abstract Kripke structure is updated. For each state $\hat{s}$ that was split into $\hat{s}_1$ and $\hat{s}_2$, the following steps are done.

- $\hat{S} \leftarrow \hat{S} \setminus \{\hat{s}\} \cup \{\hat{s}_1, \hat{s}_2\}$.
- Successors/predecessors of $\hat{s}_1$ and $\hat{s}_2$ can be queried from an SMT solver. However, only those successors/predecessors have to be considered that were already successors/predecessors of $\hat{s}$.
- $\hat{s}_1 \in \hat{I}$ and $\hat{s}_2 \in \hat{I}$ can also be queried from an SMT solver, but this check only has to be done if $\hat{s} \in \hat{I}$. Otherwise $\hat{s}_1, \hat{s}_2 \notin \hat{I}$.

4.4.5 Optimization: incremental model checking

As argued in the previous section, only a subset of the abstract states is refined using the interpolants. Let $\hat{s}_s$ denote the first state of the abstract counterexample $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_{f-1}$, $\hat{s}_f$, $\hat{s}_{f+1}, \ldots, \hat{s}_n$ that was split, i.e.,

- $\hat{s}_s = \hat{s}_f$ with Craig interpolation,
- $\hat{s}_s = \hat{s}_i$ with sequence interpolation, where i is the first index such that $\mathscr{I}_i \neq \top$ and $\mathscr{I}_i \neq \bot$.

A non-incremental model checker loops through each initial state and explores the set of reachable states. If a state violating the safety property is found, the actual path is returned. However, the search is completely restarted in the next model checking iteration despite the fact that only a subset of the states were split. The main idea of the incremental approach is presented in Figure 4.15. The path $(\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4)$ represents the actual counterexample, where $\hat{s}_3$ was split. Each state has some successors that were already fully explored (drawn on the left side of the state) and also some successors yet to be explored (drawn on the right side). There can also be initial states that were already fully explored ($\hat{s}_5$ in the figure) and initial states that will be explored after $\hat{s}_1$ ($\hat{s}_6$ in the figure). States in the gray area were fully explored before $\hat{s}_3$. Let this set be denoted by $\hat{G}$. It is clear that $\hat{s}_3$ can only be reached from $\hat{G}$ through $\hat{s}_2$. Otherwise, $\hat{s}_3$ would first be reached that way and not through $\hat{s}_2$. Therefore, splitting $\hat{s}_3$ does not affect states in $\hat{G}$. If exploration is continued with $(\hat{s}_1, \hat{s}_2)$ on the stack, $\hat{s}_2$ will "represent" states in $\hat{G}$, i.e., if some of the new states could be reached from $\hat{G}$, it will be reached from $\hat{s}_2$. Therefore, if $\hat{s}_s$ is the first state that was split ($\hat{s}_3$ in the example), states explored before $\hat{s}_s$ do not need to be re-explored and the actual path can be kept until $\hat{s}_{s-1}$.

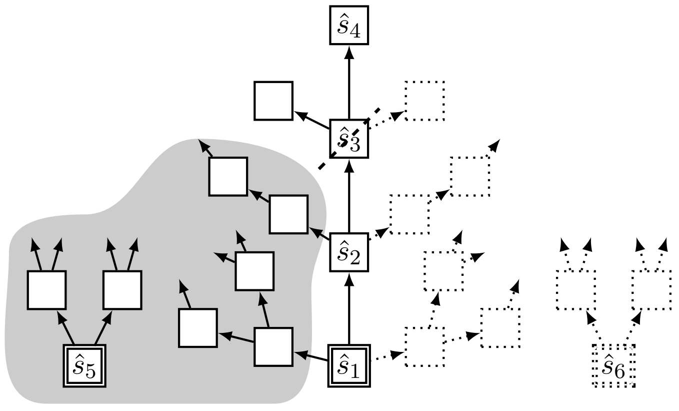


Figure 4.15: *Illustration of incremental model checking.*

The incremental model checking approach is formalized in Algorithm 6. The input of the algorithm is a symbolic transition system T, an abstract Kripke structure $\hat{M}$, the first state $\hat{s}_s$ in the counterexample that was split and a requirement $\mathsf{AG}\ \varphi$. There are some technical assumptions about the representation of the abstract Kripke structure. Initial states $\hat{I}$ are stored in a list and each state $\hat{s} \in \hat{S}$ has a list of its successors. When updating the Kripke structure during the refinement, the refined states replace the original ones in these lists. Furthermore, it is assumed that the set of explored states E, the backtrack stack Q and the index of the actual initial state i is kept between two calls.

If the model checker is called for the first time, E is initialized to be empty and i corresponds to the first initial state. Otherwise $\hat{s}_s$ and states explored after $\hat{s}_s$ are removed from both E and Q, and a flag (*isContinuation*) is set to indicate that the search is continued and some steps do not have to be repeated. The outer loop does the exploration from each initial state. The stack Q stores pairs of states and indices $(\hat{s}, k)$. The index k means that when backtracking to $\hat{s}$, the kth successor has to be explored next, i.e., the first $k - 1$

successors are already fully explored. The initialization between lines 12 and 16 only have to be done if the search is not a continuation or an initial state was split. The inner loop takes the last element of the stack Q and checks if $\hat{s} \not\models \varphi$ holds. If yes, the actual path (stored in Q) is returned as a counterexample. Otherwise the next successor of $\hat{s}$ is put on the stack. If $\hat{s}$ has no successors, it is removed from the stack for backtracking.

Algorithm 6: Check abstract state space (interpolating CEGAR).

```
Input  : $T$: Symbolic transition system
         $\hat{M}$: abstract Kripke structure of $T$
         $\hat{s}_s$: first split state
         AG $\varphi$: requirement
Output : “Requirement holds” or an abstract counterexample
1  if first iteration then
2      $E \leftarrow \emptyset$;
3      $i \leftarrow 1$;
4      $isContinuation \leftarrow$ false;
5  else
6      Remove $\hat{s}_s$ and states after $\hat{s}_s$ from $E$ and $Q$;
7      $isContinuation \leftarrow$ true;
8  end
9  while $i \leq$ size of $\hat{I}$ do
10     if $\hat{s}_s \in \hat{I} \vee \neg isContinuation$ then
11         $\hat{s}_0 \leftarrow i$th element of $\hat{I}$;
12         if $\hat{s}_0 \in E$ then continue;
13         $Q \leftarrow \{(\hat{s}_0, 1)\}$;
14         $E \leftarrow E \cup \{\hat{s}_0\}$;
15         $isContinuation \leftarrow$ false;
16     end
17     while $Q \neq \emptyset$ do
18         $(\hat{s}, k) \leftarrow$ top$(Q)$;
19         if $\neg$first iteration $\wedge\ \hat{s}_s \notin \hat{I}$ then $k \leftarrow k - 1$;
20         if $\hat{s} \in E$ then continue;
21         $E \leftarrow E \cup \{\hat{s}\}$;
22         if $\hat{s} \not\models \varphi$ then return $Q$;
23         if $k >$ number of successors of $\hat{s}$ then pop$(Q)$;
24         else
25             $\hat{s}' \leftarrow k$th successor of $\hat{s}$;
26             $k \leftarrow k + 1$;
27             push$(Q, (\hat{s}', 1))$;
28         end
29     end
30     $i \leftarrow i + 1$;
31 end
32 return “Requirement holds”
```

4.5 Summary

This section summarizes the common and different aspects of the three CEGAR approaches presented in this chapter. A short overview can be seen in Table 4.2.

Table 4.2: *Summary of the CEGAR algorithms.*

	Clustered	**Visibility-based**	**Interpolating**
Main idea	Composite abstraction functions, clustering, predicate abstraction	Visible and invisible variables	Predicate abstraction and explicitly tracked variables
Init. abs.	Clustering, create abstract Kripke structures of clusters	Collect visible variables	Create abstract Kripke structure
Model checking	On-the-fly composition, check safety property	On-the-fly exploration, check safety property	Explore constructed state space incrementally, check safety property
Counterex. examin.	Unfold	Unfold	Unfold and check last state
Abs. ref.	Split the components of the failure state	Make new variables visible based on interpolation	Lazy abstraction: split failure state with Craig interpolation or the counterexample with sequence interpolation

The clustered approach is based on composite abstraction functions and predicate abstraction, while visibility-based CEGAR uses visible and invisible variables. Interpolating CEGAR can be regarded as a combination of the former approaches: it can handle both explicitly tracked variables and predicates.

Interpolating CEGAR builds the initial Kripke structure explicitly, since it is expected to be small. In contrast, the clustered approach only constructs the component Kripke structures and the visibility-based algorithm only collects the visible variables.

Therefore, model checking is done on-the-fly in the clustered and visibility-based approaches. In contrast, the interpolating algorithm only traverses the previously built state space with an incremental optimization. All three approaches currently work on safety properties.

Examining the counterexample is performed using unfolding in all three approaches. However, the interpolating algorithm requires an extra check for the last state to violate the requirement since the model can be defined over different labels than the requirement.

Abstraction refinement is different in all three approaches. Clustered CEGAR splits the failure state by separating dead-end and bad states in its components. Visibility-based CEGAR makes some of the previously invisible variables visible, inferred from interpolants. Interpolating CEGAR either splits the failure state with a predicate from Craig interpolation or splits all states of the counterexample with predicates from sequence interpolation.

Infinite state space

The generic framework (Section 4.1) ensures termination for models with finite state space. However, the algorithms may also terminate for infinite models under some circumstances.

- The clustered approach can only deal with infinite models as long as their initial abstraction can be verified. If refinement is required, the enumeration of infinitely many dead-end states can cause non-termination.
- The visibility-based algorithm is capable of verifying infinite systems as long as the variables with infinite domains are invisible. Otherwise the abstract state space also becomes infinite. This yields non-termination if there is no counterexample, or the depth-first search chooses an infinite path, which does not contain a state violating the requirement.
- The advantage of the interpolating approach is that it does not need to enumerate concrete states or concrete values of a variable (unless it is explicitly tracked). Consequently, each step of the algorithm works on finite models. However, the refinement loop may not terminate as states may be split infinitely many times.

Contributions

The clustered algorithm is mainly based on the work of Clarke et al. [46]. However, instead of grouping concrete states, I only enumerate the abstract states using predicate abstraction [47]. In contrast to the symbolic model checker of Clarke et al., I use on-the-fly explicit model checking and I only support safety properties.

The visibility-based CEGAR is mainly based on a different work of Clarke et al. [50]. In my work however, I use Craig interpolation [21] for refinement instead of integer linear programming and decision tree learning. Furthermore, I use on-the-fly explicit model checking compared to the symbolic method of Clarke et al. [50].

The interpolating method is new approach, combining several algorithms and related techniques. It employs both predicate abstraction [47] and explicitly tracked variables (which are similar to the visibility-based method [50]). In the interpolating approach I also proposed an incremental explicit model checker. Refinement can be achieved by both Craig [62] and sequence interpolation [51], along with lazy abstraction [59].

Chapter 5

Implementation

In order to evaluate and compare the CEGAR approaches presented in Chapter 4, I implemented a prototype of each algorithm. This chapter first gives an overview on the architecture (Section 5.1), then the important features of the main components are highlighted. The main components are the TTMC framework (Section 5.2), the CEGAR core (Section 5.3) and the CEGAR algorithms (Section 5.4). Finally, Section 5.5 presents how the algorithms can be used from the command line or the graphical user interface.

5.1 Architecture

The architecture of the implementation can be seen in Figure 5.1. There are three main components: the TTMC framework, the CEGAR core and the CEGAR algorithms. All components are implemented in Java.[1]

The TTMC framework defines a metamodel and a textual language for describing symbolic transition systems. The concrete system files can be parsed into instance models that serve as an input for the algorithms. TTMC also provides a common interface for SAT/SMT solvers. This way, the underlying solver can be changed transparently. Furthermore, the framework is also equipped with a variety of utilities for manipulating formulas and systems.

The core part of CEGAR defines the base data structures that are extended with specific features by each algorithm. A generic CEGAR loop is also implemented in the core part. Each main step (initialization, checking, concretization, refinement) is an interface, which the specific algorithms must implement. There are also some core utilities that are helpful for each algorithm.

The CEGAR algorithms correspond to the three approaches presented in Chapter 4. Each algorithm has its own specific data structures and steps. The dashed line between the

[1]`http://java.com` (version 8)

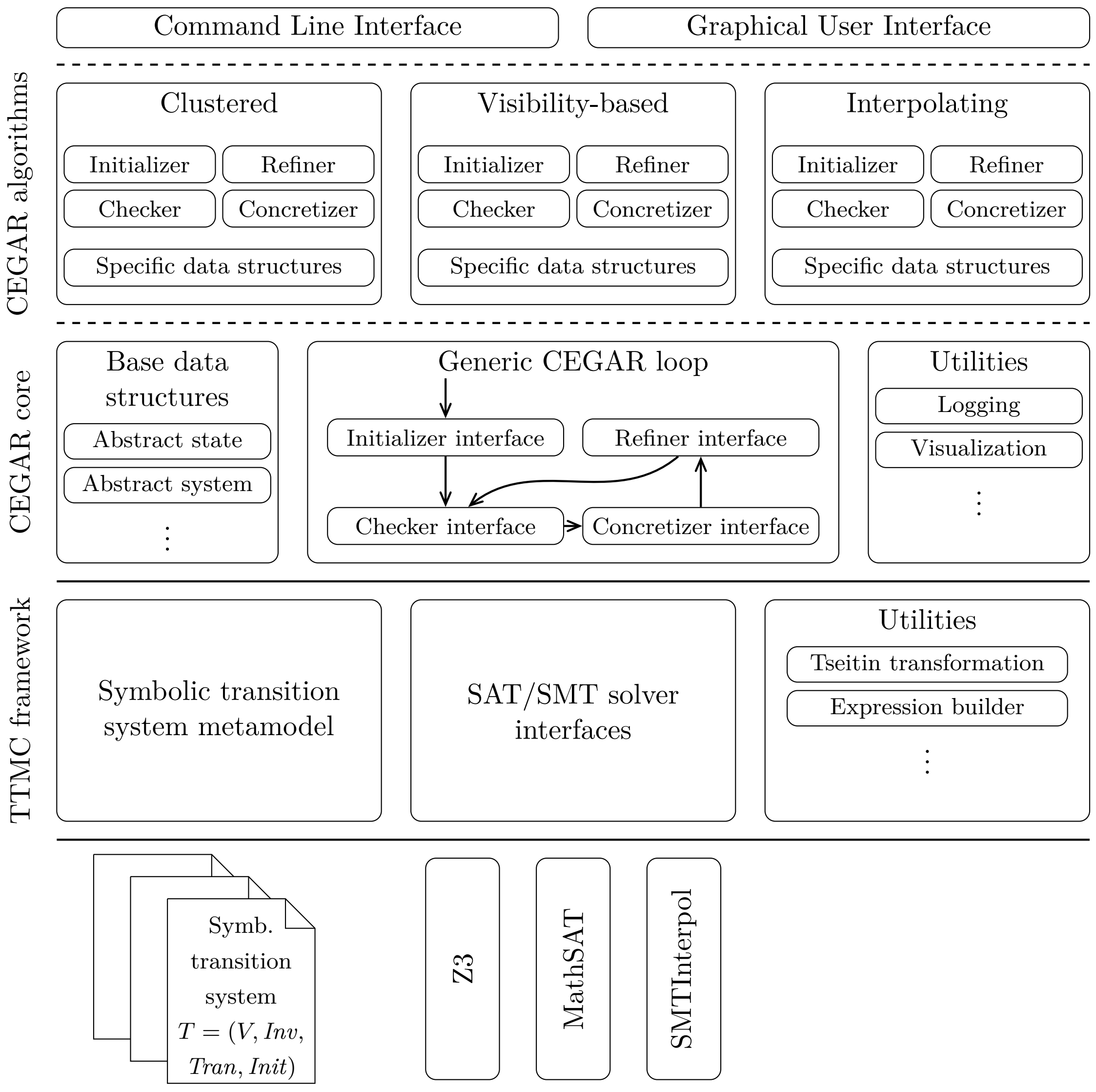


Figure 5.1: *Software architecture.*

algorithms and the core indicates a non-strict layering, i.e., the algorithms can directly access features of the framework as well.

I also implemented a command line interface (CLI) and a simple graphical user interface (GUI) to be able to deploy and run the algorithms without a development environment. The CLI and the GUI can instantiate and run the algorithms based on command line arguments or GUI elements.

5.2 TTMC framework

TTMC is the codename of a verification framework developed at the Fault Tolerant Systems Research Group of Budapest University of Technology and Economics. The framework provides a wide range of functionalities, but in my thesis I limit the discussion only to those that are actually used by the algorithms presented in this work.

Metamodel. The framework defines a textual language (e.g., Listing 2.1 on page 22) and a metamodel for describing symbolic transition systems. A parser is also included that takes a system description file and creates an instance model that serves as an input for the algorithms. A system consists of `VariableDeclarations`, `ConstraintDefinitions` and a `PropertyDeclaration`, where `ConstraintDefinitions` are `Invariant-`, `Initial-` or `TransitionConstraintDefinitions` (representing *Inv*, *Init* and *Tran* respectively), and the `PropertyDeclaration` is the requirement φ. `ConstraintDefinitions` and `Property-Declarations` contain an `Expression` that serves as a base class for each element of the logic. `Expressions` can be

- references to variables or constants, e.g., `ReferenceExpression`,
- Boolean connectives, e.g., `NotExpression`, `AndExpression`,
- functions of the logic, e.g., `AddExpression`, `SubtractExpression`,
- predicates of the logic, e.g., `EqualityExpression`, `LessExpression`,
- temporal operators, e.g., `GloballyExpression`, `UntilExpression`,
- other special expressions, e.g., `IfThenElseExpression`.

Solver interfaces. TTMC also defines a common interface for SAT/SMT solvers. This way, the concrete solver can be changed transparently to the algorithms. Currently TTMC supports MathSAT[2] [66], Z3[3] [67], and SMTInterpol[4] [68]. The latter two can also calculate interpolants. For an extensive list of SMT solvers along with their capabilities, the reader is referred to [69]. The common `TTMCSolver` interface provides the following functions:

- `Assert`: assert a collection of expressions,
- `Check`: check if the asserted expressions are satisfiable,
- `getStatus`: get the result (satisfiable or unsatisfiable),
- `getModel`: get the satisfying interpretation,
- `Push`, `Pop`: `Pop` removes expressions that were asserted after the latest `Push` (incremental solving).

Solvers that support interpolation also implement the `TTMCInterpolatingSolver` interface, which adds the following extra functions on top of `TTMCSolver`:

- `createInterpolantMarker`: create markers, e.g. α, β or $\alpha_1, \alpha_2, \dots, \alpha_{n+1}$,
- `Assert` (with marker): assert a collection of expressions for a marker,
- `getInterpolant`: get the (Craig or sequence) interpolant.

[2]`http://mathsat.fbk.eu/`
[3]`http://github.com/Z3Prover/z3`
[4]`http://ultimate.informatik.uni-freiburg.de/smtinterpol/index.html`

Utilities. The framework is also equipped with a variety of utilities for manipulating formulas and symbolic transition systems. `ExpressionBuilder` is a factory class for constructing formulas programmatically. `ExpressionExtensions` contains utility functions, for example, checking isomorphism and collecting atomic subformulas. The CNF transformation of Tseitin is also implemented as a utility function.

5.3 CEGAR core

The CEGAR core contains common functionalities shared by all CEGAR algorithms. There are two main common data structures: `IAbstractSystem` is a wrapper for an abstract transition system, while `IAbstractState` represents an abstract state. There are also some helper data structures, for example to store paths and the result of the algorithm, or to represent Kripke structures.

The most important feature of the core is the `GenericCEGARLoop`, which connects and executes the four main steps (presented in Section 4.1.2), regardless of the type of abstraction. This is implemented with the *strategy pattern* [70], i.e., each main step is an interface, for which the specific algorithms provide the implementation. These core interfaces are summarized in the following list, while their interaction can be seen in Figure 5.2.

1. `IInitializer` creates the initial abstraction with a function that takes a symbolic transition system and returns an `IAbstractSystem`.
2. `IChecker` does the model checking with a function that takes an `IAbstractSystem` and returns an `AbstractResult`, which is either a positive answer or a counterexample in a form of an `IAbstractState` list.
3. `IConcretizer` examines the counterexample with a function that takes the `IAbstractSystem` and the list of `IAbstractState`s (i.e., the counterexample). It returns a `ConcreteTrace`, which is a list of concrete states, representing either a concrete counterexample, or the longest concretizable prefix of the abstract path.
4. `IRefiner` refines the abstraction with a function that takes the `IAbstractSystem`, the list of `IAbstractState`s (i.e., the counterexample), the `ConcreteTrace` and returns the refined `IAbstractSystem`.

`GenericCEGARLoop` repeats steps 2–4 until a positive result or a concretizable counterexample is obtained.

The core also contains some utilities for logging, visualization and debugging. Logging is supported on the console (`ConsoleLogger`) or in files (`FileLogger`). The detailedness of logging can be configured by setting a minimum level. Algorithms can also visualize

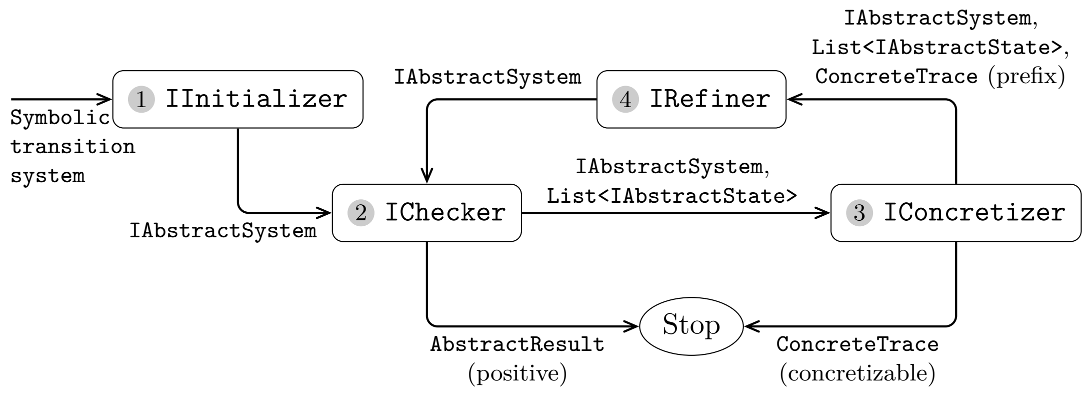

Symbolic transition system
1 IInitializer
IAbstractSystem
2 IChecker
IAbstractSystem, List<IAbstractState>
3 IConcretizer
IAbstractSystem, List<IAbstractState>, ConcreteTrace (prefix)
4 IRefiner
IAbstractSystem
Stop
AbstractResult (positive)
ConcreteTrace (concretizable)

Figure 5.2: *GenericCEGARLoop process.*

the abstract state space. Currently the format of GraphViz[5] (`GraphVizVisualizer`) and yED[6] (`YedVisualizer`) is supported.

The debugging feature (`IDebugger`) can explore and visualize the whole abstract and concrete state space, including the abstract counterexample and the concretizable prefix. Although it only works for small systems, it can be extremely useful during development and for presenting how the algorithm works. An example image produced by the debugging feature (using GraphViz) can be seen in Figure 5.3. Concrete and abstract states are denoted by ellipses and rectangles respectively. A dashed edge indicates that the state violates φ of the requirement AG φ, while a light edge means that the state is not reachable. The abstract counterexample and its concretizable part is highlighted with a background color. Initial (concrete or abstract) states are denoted by double or thick edges. Abstract states are also annotated with labels and values of (explicitly tracked and visible) variables.

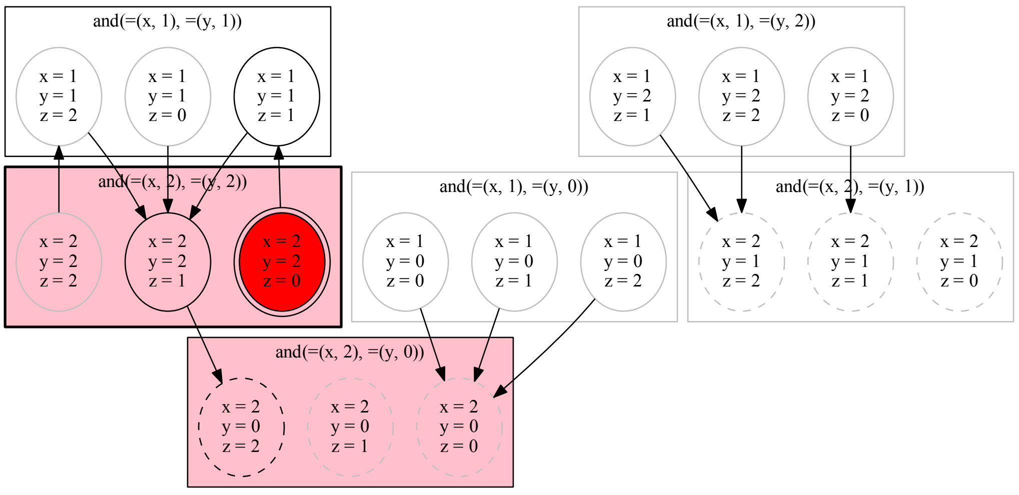


Figure 5.3: *Example image generated by the debugger.*

[5]`http://www.graphviz.org/`
[6]`http://www.yworks.com/en/products/yfiles/yed/`

5.4 CEGAR algorithms

Due to the differences in the abstraction scheme of the algorithms, each one needs to store different data about the abstract system and the abstract states. Hence, each algorithm implements and extends `IAbstractSystem` and `IAbstractState` with specific attributes.

- The abstract state of the clustered algorithm (`ClusteredAbstractState`) is a product of abstract states belonging to each cluster (`ComponentAbstractState`). Therefore, `ClusteredAbstractState` is only a collection of `ComponentAbstractState`s, while the latter one stores the predicates or their negations that hold for the state. The abstract system (`ClusteredAbstractSystem`) keeps track of the predicates, the formula and variable clusters, and the Kripke structures corresponding to the clusters.
- An abstract state in the visibility-based approach (`VisibleAbstractState`) stores the values of the visible variables. The abstract system (`VisibleAbstractSystem`) only keeps track of the visible and invisible variables. No Kripke structure is stored, since it is constructed on-the-fly during model checking.
- The abstract state in the interpolating algorithm (`InterpolatedAbstractState`) stores the values of the explicitly tracked variables and the set of predicates (or their negations) that hold for the state. The abstract system (`InterpolatedAbstract-System`) stores the initial predicates and the explicitly built Kripke structure.

Each algorithm also implements the four main steps (initialization, checking, concretization, refinement) differently. Each step is parameterized with a solver, a logger and a visualizer, but some steps also have further, algorithm specific parameters (e.g., interpolation type). A class diagram can be seen in Figure 5.4.

`ClusteredInitializer` determines the clusters and builds the initial abstract Kripke structures. `VisibleInitializer` only determines the set of visible and invisible variables. `InterpolatingInitializer` builds the initial abstract Kripke structure. It can be parameterized with the set of explicitly tracked variables and the initial predicates. It can collect initial predicates from the requirement or from the conditions of the transition relation.

Model checking is done on-the-fly in `ClusteredChecker` and `VisibleChecker`. In contrast, `InterpolatingChecker` only traverses the previously built abstract Kripke structure. Furthermore, it can be parameterized to use incremental model checking.

Concretization of the counterexample is the same for the clustered and visibility-based approaches and there is only an extra condition in the interpolating algorithm. Therefore, a base class (`ConcretizerBase`) is implemented in the core for concretizing counterexamples. `ClusteredConcretizer` and `VisibleConcretizer` calls the base class without the extra condition, while `InterpolatingConcretizer` applies the extra check as well.

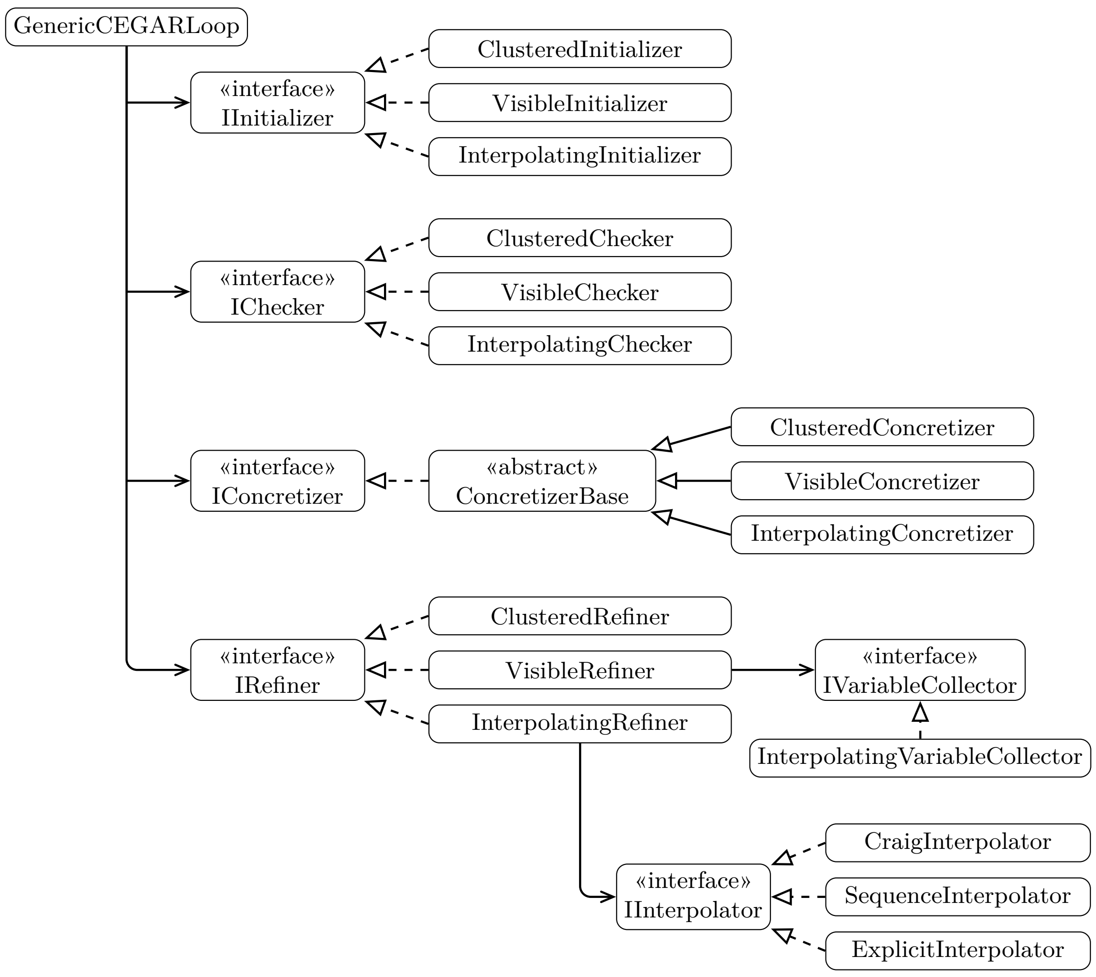


Figure 5.4: *Architecture of the main classes.*

`ClusteredRefiner` splits the components of the failure state in the component Kripke structures. `VisibleRefiner` is parameterized with an `IVariableCollector` (strategy pattern) that infers new variables to be made visible. Currently, only the interpolation based method (`InterpolatingVariableCollector`) is implemented, but the architecture is easily extensible. `InterpolatingRefiner` can be parameterized with an `IInterpolator` (strategy pattern) that calculates the interpolant. Currently, there are three interpolation methods: `CraigInterpolator`, `SequenceInterpolator` and `ExplicitInterpolator`. The first two implement Craig and sequence interpolation respectively. `ExplicitInterpolator` explicitly enumerates dead-end states into a formula, which is not efficient, but suitable for testing soundness of the other interpolation methods.

I also implemented the *builder pattern* [70] for each algorithm (`ClusteredCEGARBuilder`, `VisibleCEGARBuilder`, `InterpolatingCEGARBuilder`) to make the parameterization and instantiation of the algorithms more convenient.

5.5 Usage

The algorithms are deployed into a `jar` file, which can be run without a development environment. Both a command line and a graphical interface is provided.

Command Line Interface. The `CLI` class implements the command line interface for running the algorithms. Table 5.1 summarizes the possible arguments. The column "Alg." lists the algorithms (**C**lustered, **V**isibility-based, **I**nterpolating) for which the option is applicable.

Table 5.1: *Command line arguments.*

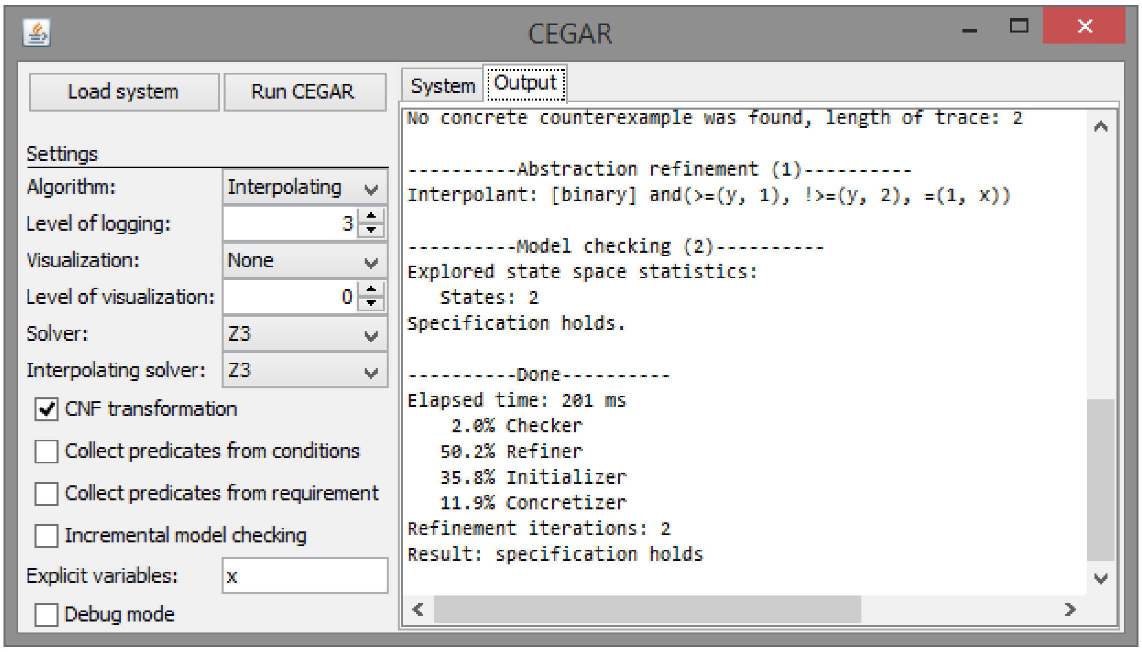
Option	Description	Alg.
`-a <algorithm>`	Algorithm to run, possible values: `clustered`, `visible`, `interpolating`	C V I
`-m <model>`	Path of the model	C V I
`-l <log>`	Logging method, possible values: `console` or a filename	C V I
`-ll <level>`	Level of logging	C V I
`-vp <path>`	Visualization path, where graphs are generated	C V I
`-vt <type>`	Visualization type, possible values: `yed`, `graphviz`	C V I
`-vl <level>`	Level of visualization	C V I
`-s <solver>`	Solver, possible values: `mathsat`, `smtinterpol`, `z3`	C V I
`-is <solver>`	Interpolating solver, possible values: `smtinterpol`, `z3`	V I
`-cnf <bool>`	Apply CNF (Tseitin) transformation	V I
`-cc <bool>`	Collect initial predicates from the conditions of *Tran*	I
`-cs <bool>`	Collect initial predicates from the requirement φ	I
`-im <method>`	Interpolation method, possible values: `craig`, `explicit`, `sequence`	I
`-imc <bool>`	Enable incremental model checking	I
`-ex <vars>`	List of explicitly tracked variables	I
`-dbg <bool>`	Enable debug mode	C V I

Parameters `-a` and `-m` are mandatory, others are optional. If an optional parameter is not given, the default value is applied.

Example 5.1. *The command below runs the interpolating algorithm on "test.system" with console logging (level 3), using the Z3 solver, applying Tseitin transformation and tracking x, y explicitly.*

```
java -jar cegar.jar   -a interpolating   -m test.system   -l console
                      -ll 3    -s z3   -is z3   -cnf true   -ex x,y
```

Graphical User Interface. If the application is started without any argument, a graphical user interface appears, which is implemented in the class `GUI`. A screenshot of the GUI can be seen in Figure 5.5. The algorithms can be configured and started with the controls on the left side of the window, while the loaded system and the output appears in the rest of the window. The algorithm runs on a background thread, therefore its output can be continuously observed (as on a console).



Settings

Algorithm: Interpolating
Level of logging: 3
Visualization: None
Level of visualization: 0
Solver: Z3
Interpolating solver: Z3
[✓] CNF transformation
[ ] Collect predicates from conditions
[ ] Collect predicates from requirement
[ ] Incremental model checking
Explicit variables: x
[ ] Debug mode

```
No concrete counterexample was found, length of trace: 2

----------Abstraction refinement (1)----------
Interpolant: [binary] and(>=(y, 1), !>=(y, 2), =(1, x))

----------Model checking (2)----------
Explored state space statistics:
   States: 2
Specification holds.

----------Done----------
Elapsed time: 201 ms
    2.0% Checker
   50.2% Refiner
   35.8% Initializer
   11.9% Concretizer
Refinement iterations: 2
Result: specification holds
```

Figure 5.5: *Graphical User Interface.*

Chapter 6

Evaluation

This chapter evaluates and compares the algorithms based on measurements for several models. Section 6.1 presents the performance of the algorithms on simple finite state models. Section 6.2 focuses on PLC models that have a finite, but large state space. Section 6.3 evaluates the algorithms on an infinite model, namely Fischer's protocol. Section 6.4 discusses the bottleneck of the algorithms identified by profiling and Section 6.5 summarizes the results. The measurements were performed with the following configuration:[1]

- Intel Core i7 4710HQ 3,5GHz processor,
- 8 GB RAM,
- Windows 8.1 x64,
- Java 8.

6.1 Simple finite state space models

This section presents the evaluation of the algorithms on simple models with finite state space. These models contain 1–4 variables and have a concrete state space between 10–100 states. Table 6.1 contains run time results for the following five configurations, corresponding to the main columns:

1. clustered,
2. visibility-based,
3. interpolating with Craig interpolation,
4. interpolating with Craig interpolation and incremental model checking,
5. interpolating with sequence interpolation and incremental model checking.

The sub-columns T, #R and #S represent the run time, the number of refinements and the sum of explored (abstract) states in each iteration respectively. The ✓ or × sign before the name of a model indicates whether it meets the requirement or not.

[1]The measurements were evaluated in a new, prototype version of the TTMC framework, which currently only supports the Z3 solver.

The *"loop"* models correspond to simple programs containing a while loop. The *"bool"* and *"simple"* models are similar to the model in Listing 4.1. As their name suggests, the *"counter"* models represent counters. The *"read_write"* model was translated from a simple Petri net, representing a resource that can be accessed by readers and writers.

Table 6.1: *Measurement results for simple finite state models.*

	Clustered			Visible			Int. (Cr.)			Int. (Cr., inc.)			Int. (seq., inc.)		
Model	T (ms)	#R	#S	T (ms)	#R	#S	T (ms)	#R	#S	T (ms)	#R	#S	T (ms)	#R	#S
✓ loop	49	0	5	38	1	20	73	5	19	56	5	14	62	5	18
× loop	56	4	40	33	1	18	208	11	78	200	11	38	327	10	62
× bool1	1	0	3	1	0	3	4	2	6	3	2	5	3	2	6
× bool2	2	0	4	3	0	8	4	3	10	4	3	7	5	3	9
✓ counter	2	0	2	6	0	11	3	1	2	2	1	2	2	1	2
× counter	26	4	20	5	0	6	30	5	21	27	5	15	40	5	18
✓ read_write	14	0	3	7	0	5	12	2	5	13	2	4	16	2	6
× simple1	15	0	3	10	0	4	8	1	3	8	1	3	9	1	3
✓ simple2	37	3	15	4	0	2	36	4	10	31	4	8	23	3	8
× simple3	38	1	7	18	1	6	40	4	12	39	4	10	21	2	7

Measurements show that for small models the visibility-based approach performs best in most cases. Usually, all variables are visible initially or after the first iteration, so the visibility-based algorithm simply enumerates the concrete states, which is efficient for models with a small concrete state space. The clustered method also performs well for these models, however, it may require more iterations to find a counterexample (e.g., *"loop"* or *"counter"*). The interpolating configurations start with a predicate set $\mathcal{P} = \{\top\}$, hence the state space exploration is guided completely by the interpolants. This yields an overhead in the run time for most models. It can be seen that the total number of explored states is less if incremental model checking is used. However, since models are small, the effect of incremental model checking on run time is also small. It can also be seen that sequence interpolation yields equal or less refinements, but it has to explore more states.

Determining the initial set of predicates

Table 6.2 compares four different strategies for determining the initial set of predicates $\mathcal{P}$ in the interpolating algorithm:

1. $\mathcal{P} = \{\top\}$,
2. $\mathcal{P} = \mathsf{atoms}(\mathit{Tran})$,
3. $\mathcal{P} = \mathsf{atoms}(\varphi)$,
4. $\mathcal{P} = \mathsf{atoms}(\mathit{Tran}) \cup \mathsf{atoms}(\varphi)$,

where $\mathsf{atoms}(\mathit{Tran})$ denotes the atomic subformulas of the conditions in the transition relation and $\mathsf{atoms}(\varphi)$ denotes the atomic subformulas of the requirement.

Measurements show that collecting initial predicates from the requirement (φ) yields less iterations and shorter run time in many cases. However, collecting predicates from the

Table 6.2: *Measurement results for predicate collecting strategies.*

	Int.			Int. (*Tran*)			Int. (φ)			Int. (*Tran*, φ)		
Model	T (ms)	#R	#S	T (ms)	#R	#S	T (ms)	#R	#S	T (ms)	#R	#S
✓ loop	99	5	14	245	11	51	57	4	13	96	0	5
× loop	185	11	40	192	9	48	182	10	39	190	9	48
× bool1	3	2	5	4	2	5	1	0	3	2	0	3
× bool2	4	3	7	4	3	7	6	0	8	6	0	8
✓ counter	2	1	2	97	10	30	1	0	1	3	0	2
× counter	28	5	15	26	4	14	25	4	14	26	4	14
✓ read_write	11	2	4	11	2	4	12	0	3	14	0	3
× simple1	25	1	3	97	0	2	20	0	2	94	0	2
✓ simple2	31	4	8	109	3	11	26	1	4	110	3	11
× simple3	35	4	10	220	1	6	41	2	8	214	1	6

conditions (*Tran*) has a negative effect on performance. If predicates are collected both from the conditions and the requirement, less iterations are required, but the run time of the algorithm is longer due to the construction of the (often unnecessarily) large initial abstract state space.

6.2 CERN PLC models

CERN, the European Organization for Nuclear Research[2] is a particle physics laboratory near Geneva, Switzerland. Physicists, engineers and computer scientists are working together at CERN to discover the fundamental structure of the universe. The main instruments built at CERN are particle accelerators and detectors. Accelerators boost beams of particles to high energies and make them collide with another beam or a fixed target. Detectors record the data at such interaction points for further analysis. Many systems at CERN use PLCs (Programmable Logic Controllers) as industrial controllers, including the LHC (Large Hadron Collider), which is the most powerful particle accelerator to date. A bug in the controller of such powerful instruments can cause serious injuries and damages.

A group at CERN is working on the formal verification of PLC programs [71]. They translate the PLC codes into an automaton-based *intermediate model*. Several reductions are applied directly on the intermediate model [72] before it is transformed to the syntax of a model checker. The intermediate model can be transformed into a symbolic transition system described in the language of the TTMC framework[3] as well, which can then be checked using the algorithms presented in this thesis. Table 6.3 contains run time, refinement iterations and the number of explored states, while Table 6.4 breaks down the run time (in percentages) to the four main steps (In.: initialization, Ch.: model checking, Co.: counterexample concretization, Re.: refinement). Where no result is available, the algorithm ran out of memory or could not verify the instance in 60 minutes.

All models were extracted from a PLC module called `OnOff` [71] based on different requirements. `OnOff` is a generic actuator module that can represent valves, heaters, motors, etc.

[2]`http://home.cern/`

[3]The transformation was implemented by Dániel Darvas and the author.

in a PLC code. The models contain 24 to 83 variables and have a potential concrete state space between 10^8 and 10^{26} states.[4]

Table 6.3: *Measurement results for PLC models.*

	Int. (sequence)			Int. (Craig)			Visible		
Model	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S
× LOCAL_vc1	56.3	34	191	27.7	33	100	41.1	7	1640
× LOCAL_vc2	55.3	34	191	29.3	33	100	36.2	7	1697
✓ REQ_1-1	117.7	23	292	218.3	109	2419	8.3	1	369
✓ REQ_1-8	15.7	16	82	46.5	64	1076	3.5	1	165
× REQ_1-8	444.6	31	1198	52.6	65	1069	8.3	2	274
✓ REQ_1-9	24.8	17	98	51.2	63	1130	3.6	1	167
✓ REQ_2-3b	–	–	–	1 663.2	159	4752	1 362.4	3	20893
✓ REQ_3-1	515.0	66	1047	224.1	58	552	84.7	2	1163
× REQ_3-2	–	–	–	104.4	37	111	75.8	2	628
✓ UCPC-1721	105.4	32	633	114.5	90	1716	15.4	5	1506

Table 6.4: *Run time percentage of each step for PLC models.*

	Int. (sequence)				Int. (Craig)				Visible			
Model	In.	Ch.	Co.	Re.	In.	Ch.	Co.	Re.	In.	Ch.	Co.	Re.
× LOCAL_vc1	0.5%	0.1%	19.1%	80.2%	1.2%	0.2%	46.5%	52.1%	0.5%	44.4%	33.2%	21.8%
× LOCAL_vc2	0.5%	0.1%	18.2%	81.2%	1.1%	0.2%	49.1%	49.7%	0.5%	54.0%	28.7%	16.8%
✓ REQ_1-1	0.2%	0.1%	4.3%	95.4%	0.1%	0.5%	42.2%	57.2%	2.4%	48.4%	24.9%	24.3%
✓ REQ_1-8	1.8%	0.1%	18.8%	79.2%	0.6%	0.6%	39.4%	59.4%	5.6%	52.8%	13.4%	28.2%
× REQ_1-8	0.1%	0.1%	2.4%	97.5%	0.5%	0.6%	42.5%	56.4%	2.4%	35.3%	42.9%	19.4%
✓ REQ_1-9	1.2%	0.1%	16.3%	82.4%	0.6%	0.8%	31.6%	67.0%	5.0%	51.7%	14.8%	28.5%
✓ REQ_2-3b	–	–	–	–	0.1%	0.3%	32.6%	67.0%	0.0%	53.8%	43.0%	3.2%
✓ REQ_3-1	0.2%	0.1%	11.7%	88.0%	0.4%	0.3%	42.9%	56.4%	0.7%	42.6%	32.8%	23.9%
× REQ_3-2	–	–	–	–	0.8%	0.1%	48.2%	50.9%	0.8%	27.5%	41.5%	30.2%
✓ UCPC-1721	0.2%	0.2%	7.4%	92.2%	0.2%	0.5%	35.0%	64.2%	1.2%	80.1%	9.0%	9.7%

The clustered algorithm was not able to verify any of the instances due to the large number of dead-end states in “PolyRefine” (Algorithm 3). The interpolating and visibility-based algorithms however, show diverse results. The visibility-based algorithm has the best performance for most of the models, except *“LOCAL_vc1”* and *“LOCAL_vc2”*, where the interpolating approach dominates. It can be observed that for most models Craig interpolation gives a better performance for a violated requirement and sequence interpolation for a fulfilled requirement. Further measurements pointed out that the bottleneck of the algorithm lies in the transformation of formulas (see Section 6.4), which has the biggest impact on refinement (especially interpolation). The interpolating algorithms refine the state space in many small steps (many iterations), in contrast to the visibility-based approach, which performs fewer, but bigger steps. The results in Table 6.4 also confirm that refinement dominates the run time of the interpolating approaches, especially sequence interpolation, while the visibility-based algorithm is mostly dominated by the model checking step. Consequently, the visibility-based algorithm is less affected by the bottleneck.

[4]The potential state space is estimated by all possible evaluations of the system variables. However, the actual state space may be smaller, as not every evaluation may be reachable.

Explicitly tracked variables

The PLC models contain a variable `loc`, which represents the location in the control flow automaton (similarly to program counters). I observed that interpolants often correspond to this variable, e.g. $loc \doteq 0$, $loc \doteq 1$, …, $loc \doteq n$. With this extra knowledge, I configured the interpolating algorithms to track this location variable explicitly. The results can be seen in Table 6.5. The columns "Int. (seq.)" and "Int. (Cr.)" contain the same results as the corresponding columns in Table 6.3. The other two columns represent the configurations where the location variable was explicitly ("expl.") tracked.

Table 6.5: *Measurement results for PLC models with explicitly-tracked variables.*

	Int. (seq.)			Int. (seq., expl.)			Int. (Cr.)			Int. (Cr., expl.)		
Model	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S
× LOCAL_vc1	56.3	34	191	52.5	1	81	27.7	33	100	46.9	20	452
× LOCAL_vc2	55.3	34	191	50.7	1	81	29.3	33	100	44.2	20	452
✓ REQ_1-1	117.7	23	292	55.9	6	212	218.3	109	2419	31.9	34	629
✓ REQ_1-8	15.7	16	82	6.6	2	47	46.5	64	1076	19.1	21	353
× REQ_1-8	444.6	31	1198	38.8	5	192	52.6	65	1069	38.6	35	650
✓ REQ_1-9	24.8	17	98	6.7	2	47	51.2	63	1130	19.5	21	352
✓ REQ_2-3b	–	–	–	238.3	2	120	1 663.2	159	4752	370.4	52	1369
✓ REQ_3-1	515.0	66	1047	171.3	1	70	224.1	58	552	179.5	26	657
× REQ_3-2	–	–	–	49.0	0	43	104.4	37	111	48.2	0	43
✓ UCPC-1721	105.4	32	633	37.5	11	193	114.5	90	1716	72.7	94	1845

In most cases the performance of the algorithms is 2–4 times or even more faster with the explicitly tracked variable, except for the *"LOCAL_vc1"* and *"LOCAL_vc2"* models. There are two models (*"REQ_2-3b"* and *"REQ_3-2"*), which could not be verified with sequence interpolation previously. However, using explicitly tracked variables the run time is even shorter than the run time of the visibility-based algorithm (see Table 6.3). It can also be seen that in most cases sequence interpolation performs better for violated requirements, while Craig interpolation for fulfilled requirements.

Note, that the idea of tracking the location variable explicitly can not only be applied to the models above. It can be generalized to any model that is derived from an automaton-based or state machine model. Moreover, even a simple heuristic can detect such variables by searching for predicates of the form $v_i \doteq 0$, $v_i \doteq 1$, …, $v_i \doteq n$ in the formulas of the symbolic transition system.

6.3 Fischer's protocol

Fischer's protocol [73] is a mutual exclusion algorithm for arbitrary many components. Mutual exclusion is achieved by a lock that can be read and written with respect to a lower and an upper time bound. The CEGAR algorithms presented in this thesis cannot handle parametric problems, therefore I ran the algorithms for a fixed number of participants. However, even a non-parametric model has an infinite state space due to the clock variables (having domain $\mathbb{Q}$). The results are summarized in the following list.

- The clustered algorithm cannot verify the initial abstraction and runs into an infinite loop when enumerating the infinitely many states in “PolyRefine” (Algorithm 3), due to variables with infinite domain.

- Each initially visible variable of the visibility-based approach has a finite domain. However, some variables with infinite domain become visible after a few iterations, which prevents the algorithm from termination.

- The interpolated algorithm is able to to verify the protocol for two and three participants. The results are presented in Table 6.6 and Table 6.7. It can be seen in Table 6.6 that Craig interpolation performs better than sequence interpolation and the effect of incremental model checking is also visible.

  Table 6.7 breaks down the runtime (in percentages) to the four main steps. Concretization and refinement together dominate run time in case of Craig interpolation. However, run time is dominated entirely by refinement with sequence interpolation. Further measurements pointed out that the bottleneck of refinement is the function that obtains the interpolant from the solver (see Section 6.4 for details).

Table 6.6: *Measurement results for Fischer’s protocol.*

	Int. (Cr.)			Int. (Cr., inc.)			Int. (seq., inc.)		
Model	T (s)	#R	#S	T (s)	#R	#S	T (s)	#R	#S
✓ fischer2	1.38	17	120	1.04	17	69	2.52	15	107
× fischer2	0.29	11	60	0.29	11	41	0.77	9	45
✓ fischer3	14.44	97	2442	13.97	97	998	72.86	101	1584
× fischer3	1.37	19	130	1.29	19	70	1.47	9	44

Table 6.7: *Run time percentage of each step for Fischer’s protocol.*

	Int. (Cr.)				Int. (Cr., inc.)				Int. (seq., inc.)			
Model	In.	Ch.	Co.	Re.	In.	Ch.	Co.	Re.	In.	Ch.	Co.	Re.
✓ fischer2	11.7%	2.0%	22.3%	63.8%	0.9%	1.7%	23.6%	73.8%	0.4%	1.0%	8.0%	90.6%
× fischer2	2.4%	2.8%	29.6%	65.2%	2.1%	3.4%	29.8%	64.7%	0.8%	1.2%	10.7%	87.2%
✓ fischer3	0.1%	2.1%	39.2%	58.5%	0.1%	0.9%	40.3%	58.7%	0.0%	0.3%	10.9%	88.8%
× fischer3	0.9%	0.7%	34.7%	63.5%	0.9%	0.5%	34.5%	64.0%	0.8%	0.2%	14.9%	84.2%

6.4 Profiling

The algorithm provides information about the distribution of run time among the four main steps (e.g., see Table 6.7). However, I also examined some (mainly long) runs of the algorithm with the JVM Monitor[5] profiling tool to get more detailed information. It turned out that in most cases the bottleneck of the algorithm is the transformation of formulas.

The formulas *Init*, *Tran*, *Inv*, φ and the labels of the states correspond to variables without indices (e.g., x, y and x', y'). However, formulas are often asserted to multiple states at the

[5]http://jvmmonitor.org/

same time (e.g., describing paths). In such cases the states are differentiated with variable indices (e.g., x_0, y_0 corresponds to s_0; x_1, y_1 corresponds to $s_1, \dots$). To demonstrate the issue, recall Example 2.13 on page 23. In order to describe concrete paths with length n, the formulas *Tran* and *Inv* have to be copied $n-1$ and n times respectively. Each variable $v \in V$ in the ith copy is replaced by v_i, while the primed version v' is replaced by v_{i+1}. This issue can also be observed in the opposite direction: the result of the solver may contain n instances of a variable, which has to be transformed back to a path of length n (see Example 2.13). When the result of the solver is a Craig or sequence interpolant, it also corresponds to one or more states. Such formulas also have to be transformed back (see Example 4.21 on page 65).

I observed that these transformations are performed mostly during concretization and refinement (especially interpolation). Profiling results show that it can be responsible for even 50–70% of the run time for models with large formulas (e.g., *"fischer3"*). Solving this issue is not trivial: a possible solution would be to use caching to store the replaced formulas. However, counterexamples often have a length of 100–1000 or even more, which means that many versions of a formula have to be stored. Also, if multiple solvers have to be supported, the formulas must be described with solver independent objects that again need to be transformed.

6.5 Summary

Measurement results show that the clustered and visibility-based approaches perform better on small, finite state space models. For larger models, the visibility-based and interpolating algorithms are more efficient. The extra knowledge of tracking a variable explicitly can also improve the performance of the interpolating approach. It can be observed that sequence interpolation is more efficient for violated requirements and Craig interpolation for fulfilled requirements. Furthermore, the interpolating algorithm was also able to verify a model with infinite state space. Profiling results identified the bottleneck of the algorithms, namely the transformation of formulas between the algorithms and the solver. The biggest impact of this bottleneck can be observed during concretization and refinement (especially interpolation).

Chapter 7

Conclusion

Counterexample-guided abstraction refinement is a promising formal verification technique that has been widely studied in the past fifteen years. In this thesis I examined, developed and evaluated three CEGAR-based algorithms under the common framework of existential abstraction. The results of this thesis have both a theoretical and a practical importance. From the theoretical point of view, I examined the background and literature of CEGAR-based model checking along with related techniques. I presented the clustered and visibility-based CEGAR approaches that are mainly based on existing algorithms from the literature. I also proposed a new algorithm, which is a combination of the former two methods and various related techniques, including lazy abstraction and interpolation.

From the practical point of view, I implemented the three algorithms in the TTMC framework and I evaluated and compared their performance. The architecture of the software provides an easily configurable and extensible framework for CEGAR-based techniques. Measurements show that the clustered and visibility-based approaches perform better on smaller (finite) models, while the visibility-based and interpolating methods are efficient for large state spaces. Moreover, the latter one is also capable of verifying infinite systems in some cases.

During my preparation of this thesis I successfully completed all of the specified objectives.

- I presented the CEGAR approach and related techniques for the model checking of transition systems (Chapter 3 and Chapter 4).
- I implemented three concrete, CEGAR-based algorithms in the TTMC framework (Chapter 5).
- I compared and evaluated the performance of the algorithms (Chapter 6),
- I proposed a new algorithm, based on interpolation, which is a combination and improvement of various algorithms and techniques (Section 4.4).

Future work. Even though all the objectives were met, there are still several opportunities for further research and improvements.

- A possible future direction is to extend the algorithms to handle different temporal formulas than only safety properties. The general framework supports full ACTL* in theory, but then an ACTL* model checker is required and the refinement step also has to deal with complex counterexamples (e.g., loops, trees).
- A new refinement strategy could be developed for the visibility-based algorithm using unsat cores: variables included in formulas of the unsat core would be made visible.
- Although the interpolating approach can handle both explicitly tracked variables and predicates at the initial abstraction, it only uses new predicates for the refinement. It would be interesting to extend the set of explicitly tracked variables during refinement.
- As argued in Chapter 3, CEGAR is a general approach that can work with other formalisms than transition systems. A possible future direction is to extend CEGAR for programs and timed automata.
- Profiling results pointed out that a bottleneck of the algorithm is the transformation of formulas. A future challenge in the implementation is the optimization of such transformations.

Acknowledgment

First of all, I would like to thank my family, especially my parents and my girlfriend for their support throughout my studies. Without their help, it would have been much more difficult to focus on my studies and researches.

The roots of this work go back to 2012, when András Vörös offered me the opportunity to work on a Petri net related CEGAR algorithm under his supervision. I am "infinitely" thankful for his help and support towards my studies, researches, publications and my participation in conferences and scholarships. I would also like to express my acknowledgment to Tamás Tóth, who joined András for the supervision of this work. I am thankful for his help and ideas both on the theoretical side and in the implementation.

Furthermore, I would like to thank Dániel Darvas for his remarks and advices on the draft version of my thesis, for providing the PLC models and for the opportunity to present my work to their group at CERN. Last but not least, I am also thankful for the ideas, remarks and comments of Vince Molnár.

List of Figures

List of Tables

Bibliography

[1] C. Baier and J.-P. Katoen, *Principles of Model Checking.* MIT Press, 2008.

[2] A. R. Bradley and Z. Manna, *The calculus of computation: Decision procedures with applications to verification.* Springer, 2007.

[3] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pp. 151–158, ACM, 1971.

[4] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon, “The international SAT solver competitions,” *AI Magazine*, vol. 33, no. 1, pp. 89–92, 2012.

[5] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving,” *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[6] J. P. Marques-Silva, K. Sakallah, *et al.*, “GRASP: A search algorithm for propositional satisfiability,” *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[7] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability.* IOS press, 2009.

[8] Y. Vizel, G. Weissenbacher, and S. Malik, “Boolean satisfiability solvers and their applications in model checking,” *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2021–2035, 2015.

[9] G. Tseitin, “On the complexity of derivation in propositional calculus,” in *Automation of Reasoning*, Symbolic Computation, pp. 466–483, Springer, 1983.

[10] D. A. Plaisted and S. Greenbaum, “A structure-preserving clause form translation,” *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, 1986.

[11] A. Church, “A note on the Entscheidungsproblem,” *The Journal of Symbolic Logic*, vol. 1, no. 1, pp. 40–41, 1936.

[12] A. M. Turing, “On computable numbers, with an application to the Entscheidungsproblem,” *Journal of Math*, vol. 58, pp. 345–363, 1936.

[13] R. Stansifer, “Presburger’s article on integer arithmetic: Remarks and translation,” tech. rep., Cornell University, 1984.

[14] K. Gödel, “Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I,” *Monatshefte für mathematik und physik*, vol. 38, no. 1, pp. 173–198, 1931.

[15] A. Schrijver, *Theory of linear and integer programming.* John Wiley & Sons, 1998.

[16] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, ch. 26, pp. 825–885, IOS press, 2009.

[17] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, “Deciding equality formulas by small domains instantiations,” in *Computer Aided Verification*, vol. 1633 of *Lecture Notes in Computer Science*, pp. 455–469, Springer, 1999.

[18] R. Sebastiani, “Lazy satisfiability modulo theories,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, pp. 141–224, 2007.

[19] L. De Moura and N. Bjørner, “Satisfiability modulo theories: Introduction and applications,” *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[20] G. Nelson and D. C. Oppen, “Simplification by cooperating decision procedures,” *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 2, pp. 245–257, 1979.

[21] K. McMillan, “Applications of Craig interpolants in model checking,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3440 of *Lecture Notes in Computer Science*, pp. 1–12, Springer, 2005.

[22] W. Craig, “Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory,” *The Journal of Symbolic Logic*, vol. 22, no. 03, pp. 269–285, 1957.

[23] G. Huang, “Constructing Craig interpolation formulas,” in *Computing and Combinatorics*, vol. 959 of *Lecture Notes in Computer Science*, pp. 181–190, Springer, 1995.

[24] A. Cimatti, A. Griggio, and R. Sebastiani, “Efficient generation of Craig interpolants in satisfiability modulo theories,” *ACM Transactions on Computational Logic*, vol. 12, no. 1, pp. 7:1–7:54, 2010.

[25] Y. Vizel and O. Grumberg, “Interpolation-sequence based model checking,” in *Formal Methods in Computer-Aided Design*, pp. 1–8, IEEE, 2009.

[26] S. A. Kripke, “Semantical analysis of modal logic i normal modal propositional calculi,” *Mathematical Logic Quarterly*, vol. 9, no. 5-6, pp. 67–96, 1963.

[27] L. Lamport, “Sometime is sometimes not never: On the temporal logic of programs,” in *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 174–185, ACM, 1980.

[28] E. A. Emerson and J. Y. Halpern, "“Sometimes” and “not never” revisited: On branching versus linear time temporal logic," *Journal of the ACM*, vol. 33, no. 1, pp. 151–178, 1986.

[29] E. Clarke, "The birth of model checking," in *25 Years of Model Checking*, vol. 5000 of *Lecture Notes in Computer Science*, pp. 1–26, Springer, 2008.

[30] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs*, vol. 131 of *Lecture Notes in Computer Science*, pp. 52–71, Springer, 1982.

[31] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium on Programming*, vol. 137 of *Lecture Notes in Computer Science*, pp. 337–351, Springer, 1982.

[32] E. Clarke, S. Jha, Y. Lu, and H. Veith, "Tree-like counterexamples in model checking," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 19–29, IEEE, 2002.

[33] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking.* MIT Press, 1999.

[34] P. Schnoebelen, "The complexity of temporal logic model checking," *Advances in modal logic*, vol. 4, no. 35, pp. 393–436, 2002.

[35] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Logics for Concurrency*, vol. 1043 of *Lecture Notes in Computer Science*, pp. 238–266, Springer, 1996.

[36] D. Peled, "All from one, one for all: On model checking using representatives," in *Computer Aided Verification*, vol. 697 of *Lecture Notes in Computer Science*, pp. 409–423, Springer, 1993.

[37] P. Godefroid, D. Peled, and M. Staskauskas, "Using partial-order methods in the formal validation of industrial concurrent programs," *IEEE Transactions on Software Engineering*, vol. 22, no. 7, pp. 496–507, 1996.

[38] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10^{20} states and beyond," in *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pp. 428–439, IEEE, 1990.

[39] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[40] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1579 of *Lecture Notes in Computer Science*, pp. 193–207, Springer, 1999.

[41] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, 1994.

[42] W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi, "Tearing based automatic abstraction for CTL model checking," in *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, pp. 76–81, IEEE, 1997.

[43] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 238–252, ACM, 1977.

[44] G. Bruns and P. Godefroid, "Model checking partial state spaces with 3-valued temporal logics," in *Computer Aided Verification*, vol. 1633 of *Lecture Notes in Computer Science*, pp. 274–287, Springer, 1999.

[45] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, vol. 1855 of *Lecture Notes in Computer Science*, pp. 154–169, Springer, 2000.

[46] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[47] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, vol. 1254 of *Lecture Notes in Computer Science*, pp. 72–83, Springer, 1997.

[48] E. Ermis, J. Hoenicke, and A. Podelski, "Splitting via interpolants," in *Verification, Model Checking, and Abstract Interpretation*, vol. 7148 of *Lecture Notes in Computer Science*, pp. 186–201, Springer, 2012.

[49] M. Leucker, G. Markin, and M. Neuhäußer, "A new refinement strategy for CEGAR-based industrial model checking," in *Hardware and Software: Verification and Testing*, vol. 9434 of *Lecture Notes in Computer Science*, pp. 155–170, Springer, 2015.

[50] E. M. Clarke, A. Gupta, and O. Strichman, "SAT-based counterexample-guided abstraction refinement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 7, pp. 1113–1123, 2004.

[51] D. Beyer and S. Löwe, "Explicit-state software model checking based on CEGAR and interpolation," in *Fundamental Approaches to Software Engineering*, vol. 7793 of *Lecture Notes in Computer Science*, pp. 146–162, Springer, 2013.

[52] C. Tian, Z. Duan, and Z. Duan, "Making CEGAR more efficient in software model checking," *IEEE Transactions on Software Engineering*, vol. 40, no. 12, pp. 1206–1223, 2014.

[53] Y. Vizel, O. Grumberg, and S. Shoham, “Intertwined forward-backward reachability analysis using interpolants,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 7795 of *Lecture Notes in Computer Science*, pp. 308–323, Springer, 2013.

[54] D. Beyer, M. Dangl, and P. Wendler, “Combining k-induction with continuously-refined invariants,” *arXiv preprint arXiv:1502.00096*, 2015.

[55] K. L. McMillan, “Lazy abstraction with interpolants,” in *Computer Aided Verification*, vol. 4144 of *Lecture Notes in Computer Science*, pp. 123–136, Springer, 2006.

[56] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald, “Abstraction and counterexample-guided refinement in model checking of hybrid systems,” *International Journal of Foundations of Computer Science*, vol. 14, no. 4, pp. 583–604, 2003.

[57] A. Hajdu, A. Vörös, and T. Bartha, “New search strategies for the Petri net CEGAR approach,” in *Application and Theory of Petri Nets and Concurrency*, vol. 9115 of *Lecture Notes in Computer Science*, pp. 309–328, Springer, 2015.

[58] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder, “Tutorial on parameterized model checking of fault-tolerant distributed algorithms,” in *Formal Methods for Executable Software Models*, vol. 8483 of *Lecture Notes in Computer Science*, pp. 122–171, Springer, 2014.

[59] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 58–70, ACM, 2002.

[60] M. Weiser, “Program slicing,” in *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449, IEEE, 1981.

[61] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim, “Slicing abstractions,” in *International Symposium on Fundamentals of Software Engineering*, vol. 4767 of *Lecture Notes in Computer Science*, pp. 17–32, Springer, 2007.

[62] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, “Abstractions from proofs,” in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 232–244, ACM, 2004.

[63] D. Beyer, S. Löwe, and P. Wendler, “Sliced path prefixes: An effective method to enable refinement selection,” in *Formal Techniques for Distributed Objects, Components, and Systems*, vol. 9039 of *Lecture Notes in Computer Science*, pp. 228–243, Springer, 2015.

[64] K. L. McMillan, *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[65] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Data structures for disjoint sets,” in *Introduction to Algorithms*, ch. 21, pp. 498–524, MIT press, 2001.

[66] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, “The MathSAT5 SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 7795 of *Lecture Notes in Computer Science*, pp. 93–107, Springer, 2013.

[67] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[68] J. Christ, J. Hoenicke, and A. Nutz, “SMTInterpol: An interpolating SMT solver,” in *Model Checking Software*, vol. 7385 of *Lecture Notes in Computer Science*, pp. 248–254, Springer, 2012.

[69] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump, “6 years of SMT-COMP,” *Journal of Automated Reasoning*, vol. 50, no. 3, pp. 243–277, 2013.

[70] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software.* Pearson Education, 1994.

[71] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. González Suárez, “Applying model checking to industrial-sized PLC programs,” *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[72] D. Darvas, B. Fernández Adiego, A. Vörös, T. Bartha, E. Blanco Viñuela, and V. M. González Suárez, “Formal verification of complex properties on PLC programs,” in *Formal Techniques for Distributed Objects, Components, and Systems*, vol. 8461 of *Lecture Notes in Computer Science*, pp. 284–299, Springer, 2014.

[73] B. Dutertre, M. Sorea, *et al.*, “Timed systems in SAL,” tech. rep., SRI International, Computer Science Laboratory, 2004.

Appendix

A.1 Notations

$\bot$	False	(page 10)
$\top$	True	(page 10)
a, b, c	FOL constants	(page 14)
$\alpha_{\mathcal{I}}$	Assignment of an interpretation	(page 15)
α, β	Formulas for interpolation	(page 19)
A	Atomic propositions	(page 20)
A	Universal path quantifier	(page 24)
$\mathcal{A}$	Axioms of a first order theory	(page 16)
atoms	Atomic subformulas	(page 43)
d, e	Elements of a domain	(page 21)
$D_{\mathcal{I}}$	Domain of an interpretation	(page 15)
D_v	Domain of a variable	(page 21)
E	Existential path quantifier	(page 24)
en	Encoding function (Tseitin transf.)	(page 13)
f, g, h	FOL functions	(page 14)
F	Future operator	(page 24)
FC	Formula cluster	(page 43)
G	Globally operator	(page 24)
$\hbar$	Abstraction function	(page 38)
$\hbar^{-1}$	Concretization function	(page 38)
I	Set of initial states	(page 20)
$\hat{I}$	Set of abstract initial states	(page 38)
$\mathscr{I}$	Interpolant	(page 19)
$\mathcal{I}$	Interpretation	(page 11)
$Init$	Initial condition formula	(page 21)
Inv	Invariant formula	(page 21)
L	Labeling function	(page 20)
$\hat{L}$	Abstract labeling function	(page 38)
M	Kripke structure	(page 20)
$\hat{M}$	Abstract Kripke structure	(page 38)
N	Node (of a tree)	(page 45)

p, q, r	FOL predicates	(page 14)
P, Q	Propositional variables	(page 10)
$\mathcal{P}$	Set of FOL predicates	(page 57)
φ, ψ	Formulas	(page 10)
π	Path	(page 21)
$\hat{\pi}$	Abstract path	(page 39)
proj	Projection function	(page 52)
R	Set of transitions	(page 20)
$\hat{R}$	Set of abstract transitions	(page 38)
rep	Representative function (Tseitin transf.)	(page 13)
s, t	States	(page 21)
$\hat{s}, \hat{t}$	Abstract states	(page 38)
S	Set of states	(page 20)
$\hat{S}$	Set of abstract states	(page 38)
S_B	Set of bad states	(page 50)
S_D	Set of dead-end states	(page 50)
S_I	Set of irrelevant states	(page 50)
Σ	Signature of a first order theory	(page 16)
T	Symbolic transition system	(page 21)
$\mathcal{T}$	First order theory	(page 16)
Tran	Transition formula	(page 21)
U	Until operator	(page 24)
v	Variable of a symbolic transition system	(page 21)
V	Set of variables	(page 21)
V_E	Set of explicitly tracked variables	(page 57)
V_I	Set of invisible variables	(page 53)
V_V	Set of visible variables	(page 53)
var	Variables of a formula	(page 43)
VC	Variable cluster	(page 43)
x, y, z	FOL variables	(page 14)
X	Next state operator	(page 24)

A.2 Abbreviations

BMC Bounded Model Checking

CDCL Conflict-Driven Clause Learning

CEGAR Counterexample-Guided Abstraction Refinement

CERN European Organization for Nuclear Research

CFA Control Flow Automaton

CLI Command Line Interface

CNF Conjunctive Normal Form

CTL Computational Tree Logic

DPLL Davis-Putnam-Logemann-Loveland algorithm

FOL First Order Logic

GUI Graphical User Interface

LTL Linear Temporal Logic

NNF Negation Normal Form

PL Propositional Logic

PLC Programmable Logic Controller

ROBDD Reduced Ordered Binary Decision Diagram

SAT Boolean satisfiability problem

SMT Satisfiability Modulo Theories

TTMC Timed Transition Model Checker

A.3 Index