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

Effective Domain-Specific Formal Verification Techniques

Ph.D. Dissertation

Ákos Hajdu

Thesis supervisor:
Zoltán Micskei, Ph.D. (BME)

Budapest
2020

Ákos Hajdu
https://hajduakos.github.io

June 2020

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

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

H-1117 Budapest, Magyar tudósok körútja 2.

doi: 10.5281/zenodo.3892347

Declaration of own work and references

I, Ákos Hajdu, hereby declare that this dissertation, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Hajdu Ákos kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2020.06.14.

Hajdu Ákos

Acknowledgements

This dissertation marks a checkpoint in my career, and I am grateful for so many people being a part of this incredible journey, helping me develop both professionally and personally.

First of all, I would like to thank my family for their continuous and unconditional support, and for being by my side during all the highs and lows: my wife[1] Dorka, my parents “Anya” and “Apa”, my grandparents “Mama” and “Papa” and my brother Bálint. Their continuous help and efforts made me much easier to focus on my research. I am especially grateful to Dorka for all the small things and for enduring my long times abroad when she was saying “good morning” though it was midnight.

From the professional point of view, I would like to start with my Ph.D. supervisor, Zoltán Micskei, who is not only a professional advisor but also a personal mentor for me. I am thankful for his efforts in steering every aspect of my career, including various soft skills and self-management. There was no challenge, which could not be tackled with one of his precisely organized tables. I am also grateful to Dejan Jovanović for being my supervisor during two fruitful internships at SRI International, which resulted in the third thesis of the dissertation. Discussing with him is always a pleasure: going from a high-level view all the way to the tiny details, and finally, resolving one issue yields three new interesting challenges to work on. Furthermore, I am also thankful for my former B.Sc. and M.Sc. advisors, András Vörös, Tamás Bartha and Tamás Tóth. Their role was invaluable in paving my path as a young researcher. I am also grateful for the support of professors István Majzik, Dániel Varró and András Pataricza. Despite being busy, as usual for professors, they are always happy to take some of their time and help me. I would also like to thank Ákos Kiss (University of Szeged) and Prof. Florian Zuleger (TU Wien) for their valuable time and constructive feedback. Their initial reviews, questions and suggestions helped to increase the quality of the dissertation for its final version.

I would like to thank my current and former colleagues (in no particular order) for all the discussions and collaboration: Vöri, Tomi, Vince, Gábor, Ati, Dani, Marci, Manuel, Sidi, Kristóf, Rebus, Bence, Dávid, Oszkár, Imre and Laci. I am especially thankful to Gábor for being an infrastructural and technical helpdesk. Furthermore, I am grateful to everyone at my research group ftsrg, and at our department MIT for being not only colleagues but also good friends. I am also thankful to the talented students whom I had the pleasure to work with: Gyula, Bence, Rebus, Dorina, Misi, and many others.

I was honored to work with many outstanding people during various internships, where they widened my perspective and helped my professional and personal development. Among many others, I would like to thank Zoltán Theisz (evopro), Bertrand Bellenot, Axel Naumann, the ROOT team, and the people of PH-SFT (CERN), Dejan Jovanović, Michael Emmi, Gabriela Ciocarlie and my former colleagues at CSL (SRI International) and Prof. Dániel Varró (McGill University).

Finally, I would like to thank all my friends for all the days when they picked me up when I was in pieces; for all the nights where we made memories that will never fade; for reminding me that success is my only option; for keeping me up above in my head instead of going under; and for the fact that in the end, every place felt like somewhere I belong: the wolfpack, the ski crew (a.k.a the real risk takers), the kings of VIK, skaters from EGH, the Schuman squad at CERN, my fellow SRI interns, the Long Islanders from NYC, the basketball and hike gang from Marktoberdorf, and all the people from any corner of the world whom I’ve shared some cool story with. Without listing dozens of names, I’m sure you know that I’m talking about you, and I want you all to know how thankful I am. I hope that we’ll meet again soon for more adventures. I am also grateful to Prof. Berkes, Reni, Kata and the rest of the medical staff for putting me together so that I can continue my journey. Credits also go to Fifi, the guinea pig and Pogi, the rabbit, for bringing fluffiness and joy even to the hardest working days.

[1]Technically, she was affiliated with me as my girlfriend during the work described in this dissertation.

Support. The research described in the dissertation and the publication of its results received financial support from various grants, awards, scholarships, and institutions: the MTA-BME Lendület Cyber-Physical Systems Research Group, the National Scholarship for Young Talents (NTP-NFTÖ-16 and NTP-NFTÖ-18), the R5-COP project, the European Organization for Nuclear Research (CERN), the DisCoTec 2016 student grant, the FMCAD 2017 student travel award, McGill University, SRI International, the Oberwolfach Research Institute for Mathematics (MFO), the Marktoberdorf 2016 grant, IncQuery Labs Ltd., and the Schnell László Foundation.

Summary

Formal verification techniques allow rigorous reasoning about the operation of computer systems and programs. With a sound and complete mathematical basis, it is both possible to show the presence of certain kinds of errors and to prove their absence. Formal methods are often applied in critical domains (e.g. industrial controllers) to increase quality and trust in their correct operation. However, most of the interesting questions to be analyzed are computationally complex or undecidable in general. Therefore, verification approaches in different problem domains usually put more emphasis on different properties of the analysis to achieve a reasonable trade-off. Such properties include (1) expressive power (2) efficiency, and (3) the amount of conclusive answers. This work addresses challenges related to the properties mentioned above in three different problem domains using different approaches to make verification effective.

Thesis 1 targets concurrent and asynchronous systems by modeling them with Petri nets and checking the reachability of a given state. We study an existing algorithm that uses an efficient structural over-approximation. However, the expressive power is limited to simple reachability properties and the algorithm can easily give inconclusive answers due to its iteration strategy. We lift the expressive power of the algorithm by handling a generalized version of reachability and supporting Petri nets extended with inhibitor arcs. We increase the number of conclusive answers by the algorithm via a new iteration approach on invariants and a hybrid search strategy.

Thesis 2 targets embedded software code by modeling them with control-flow automata and checking the reachability of a distinguished error location. We study abstraction-refinement-based model checking where efficiency is still a significant limitation due to the complexity of the programs and the rich domains of their variables. We propose various efficient strategies for both abstraction and refinement. For abstraction, we extend the explicit-value analysis with limited successor enumeration, and we adapt structural information from the program to guide the search more efficiently. For refinement, we develop a backward-search based interpolation strategy and an approach that uses multiple counterexamples for a faster convergence to the appropriate level of abstraction.

Thesis 3 targets decentralized, blockchain-based systems by translating contracts (programs running on such systems) to an intermediate verification language. We adapt existing modular specification constructs to this context and also propose domain-specific properties. This provides a flexible and expressive approach to specify high-level, functional properties of contracts. We define a translation from contracts to the Boogie intermediate verification language, and leverage existing modular verification approaches. Furthermore, we develop a modular encoding of arithmetic that can capture operations precisely and efficiently over large bit-widths that are common in this domain.

All contributions have been implemented in practical tools and are available publicly. We also evaluate our contributions on various synthetic and real-world examples to prove their applicability. Results show that our contributions successfully address the targeted challenges and provide effective verification approaches in general.

Összefoglaló

A formális verifikáción alapuló módszerek lehetővé teszik különböző számítógépes rendszerek és programok precíz vizsgálatát. A matematikai alapoknak köszönhetően képesek bizonyos típusú hibák megléte mellett azok hiányát is bizonyítottan igazolni. Formális módszereket gyakran kritikus környezetekben alkalmaznak (pl. ipari vezérlők) annak érdekében, hogy a rendszer minőségét és megbízhatóságát növeljék. A legtöbb formálisan vizsgálandó kérdés azonban általános esetben túl nagy számításigénnyel rendelkezik vagy elméletileg is eldönthetetlen probléma. Ennek következtében a verifikációs módszerek különböző területeken az analízis különböző tulajdonságaira fektetnek nagyobb hangsúlyt annak érdekében, hogy megfelelő kompromisszumot biztosítsanak. Ilyen tulajdonság többek között a (1) kifejezőerő (2) a hatékonyság és (3) a megválaszolt problémák száma. Jelen disszertáció az előbb felsorolt tulajdonságokhoz kapcsolódó kihívásokat vizsgál három különböző problématerületen, különböző algoritmusok segítségével annak érdekében, hogy jól alkalmazható verifikációs módszereket biztosítson.

Az 1. tézis párhuzamos és aszinkron rendszereket modellez Petri-hálók segítségével és egy adott rendszerállapot elérhetőségét ellenőrzi. A tézisben egy meglévő algoritmust vizsgálunk, amely egy hatékony, strukturális felülbecslést alkalmaz. Az algoritmus kifejezőereje azonban egyszerű elérhetőségi kérdésekre korlátozódik, továbbá az iterációs stratégiája miatt számos problémát nem tud megválaszolni. Jelen munkában kiegészítjük az algoritmust úgy, hogy az elérhetőség egy általánosított változatát és tiltó éles Petri-hálókat is támogasson, ezáltal emelve a kifejezőerejét. Emellett a megválaszolt problémák körét is kiterjesztjük egy invariánsok feletti új iterációs stratégia és egy hibrid keresési módszer segítségével.

A 2. tézis beágyazott programkódokat modellez vezérlési folyam automaták segítségével és egy kitüntetett hibaállapot elérhetőségét vizsgálja. A tézisben egy absztrakciófinomítás-alapú algoritmust vizsgálunk, amely esetén a hatékonyság továbbra is egy korlátozó tényező a programok komplexitása és a változóik gazdag értékkészlete miatt. Jelen munkában számos hatékony stratégiát javasolunk mind az absztrakció, mind a finomítás lépéseire. Az absztrakció esetén az explicit-érték analízist kiegészítjük a rákövetkező állapotok korlátolt felsorolásával és a programból származó strukturális információkat használunk fel a keresés hatékonyabb működése érdekében. A finomításhoz kidolgozunk egy hátrafelé keresésen alapuló interpolációs stratégiát és egy olyan módszert, amely több ellenpélda alapján képes finomítani. Ezáltal gyorsabb konvergenciát érünk el a megfelelő absztrakciós szint irányába.

A 3. tézis decentralizált, blokklánc-alapú rendszereket vizsgál azáltal, hogy a rajtuk futó programokat – úgynevezett szerződéseket – egy köztes verifikációs nyelvre képezi le. A meglévő moduláris specifikációs konstrukciók mellé blokklánc-specifikus tulajdonságokat is definiálunk. Ezáltal egy rugalmas és nagy kifejezőerejű módszert biztosítunk a szerződések magasszintű, funkcionális tulajdonságainak definiálására. A szerződésekről egy leképezést javasolunk a Boogie köztes verifikációs nyelvre és meglévő moduláris verifikációs módszereket használunk az ellenőrzésükre. Emellett kidolgozzuk egy olyan – maradékos osztáson alapuló – elkódolását az aritmetikai műveleteknek amely precízen és hatékonyan tudja kezelni az ezen a területen gyakori nagy bitszélességű változókat is.

A disszertáció összes kontribúciója szabadon elérhető és praktikus eszközökben került implementálásra. A kontribúciók alkalmazhatóságát számos mesterséges és valódi példán történő kiértékeléssel demonstráljuk. Az eredmények azt mutatják, hogy a kontribúcióink sikeresen célozzák meg a kitűzött kihívásokat és jól alkalmazható verifikációs módszereket biztosítanak.

List of Abbreviations

Abbreviation	Introduced in	Description
AIGER	Sec. 2.3	And-inverter graph format
ARG	Def. 2.6	Abstract reachability graph
AST	Sec. 3.3	Abstract syntax tree
BDD		Binary decision diagram
BFS		Breadth-first search
CEGAR	Sec. 1.1.4 and 2.1.3	Counterexample-guided abstraction refinement
CFA	Def. 2.3	Control-flow automaton
DFS		Depth-first search
EVM	Sec. 3.1.2	Ethereum virtual machine
FOL	Sec. 2.1.1	First-order logic
HWMCC	[Cab+16]	Hardware Model Checking Competition
ILP	Sec. 1.1.3	Integer linear programming
IVL		Intermediate verification language
LP	Sec. 1.1.3	Linear programming
LTS	Sec. 2.3	Labeled transition system
MCC	[Kor+12]	Model Checking Contest
PLC		Programmable logic controller
PN	Def. 1.1	Petri net
PS	Def. 1.5	Partial solution
RQ		Research question
SAT	Def. 2.1	Boolean satisfiability problem
SCC	Sec. 1.1.4.7	Strongly connected component
SMT	Def. 2.2	Satisfiability modulo theories
STS	Sec. 2.3	Symbolic transition system
SV-COMP	[Bey17]	Competition on Software Verification
TC	Sec. 3.2.2.7	Type condition
VC		Verification condition
XTA	Sec. 2.3	Extended timed automata

Contents

Introduction **1**
Properties and Challenges . . . 2
Problem Domains and Contributions . . . 4
Concurrent and Asynchronous Systems . . . 5
Embedded Software Code . . . 6
Blockchain-Based Decentralized Systems . . . 8

1 Extensions to the CEGAR Approach on Petri Nets **9**
1.1 Background . . . 9
1.1.1 Petri Nets . . . 9
1.1.2 Reachability Problem . . . 11
1.1.3 Linear Programming . . . 12
1.1.4 CEGAR for Petri Nets . . . 13
1.2 Extensions . . . 25
1.2.1 Reachability of Predicates . . . 25
1.2.2 Inhibitor Arcs . . . 26
1.2.3 Distant Invariants . . . 28
1.2.4 Hybrid Search . . . 34
1.3 Implementation . . . 36
1.4 Evaluation . . . 37
1.4.1 RQ1: Scalability . . . 37
1.4.2 RQ2: Comparison to Other Tools and Algorithms . . . 40
1.4.3 RQ3: Comparison of Search Strategies . . . 40
1.5 Related Work . . . 42
1.6 Summary and Future Work . . . 42

2 Efficient Strategies for CEGAR-based Software Model Checking **45**
2.1 Background . . . 45
2.1.1 Mathematical Logic . . . 45
2.1.2 Control-Flow Automata . . . 46
2.1.3 Counterexample-Guided Abstraction Refinement (CEGAR) . . . 47
2.2 Algorithmic Improvements . . . 55
2.2.1 Configurable Explicit Domain . . . 55

2.2.2 Error Location-Based Search . . . 58
2.2.3 Backward Binary Interpolation . . . 60
2.2.4 Multiple Refinements for a Counterexample . . . 61
2.3 Implementation . . . 63
2.4 Evaluation . . . 65
2.4.1 Experiment Planning . . . 65
2.4.2 Results and Analysis . . . 70
2.4.3 Comparison to Other Tools . . . 76
2.5 Related Work . . . 78
2.6 Summary and Future Work . . . 80

3 Modular Specification and Verification of Smart Contracts 83
3.1 Background . . . 83
3.1.1 Blockchain-Based Systems . . . 83
3.1.2 Ethereum . . . 84
3.1.3 Solidity . . . 85
3.1.4 Boogie IVL . . . 87
3.1.5 Modular Verification . . . 89
3.2 Modular Specification and Verification for Solidity . . . 89
3.2.1 Specification Annotations . . . 90
3.2.2 Translation . . . 92
3.3 Implementation . . . 99
3.4 Evaluation . . . 100
3.4.1 RQ1: Language Coverage . . . 101
3.4.2 RQ2: Unannotated Contracts . . . 102
3.4.3 RQ3: Annotated Contracts . . . 104
3.5 Related Work . . . 106
3.6 Summary and Future Work . . . 107

Summary of the Research Results 109
Thesis 1: Extensions to the CEGAR Approach on Petri Nets . . . 109
Thesis 2: Efficient Strategies for CEGAR-based Software Model Checking . . . 110
Thesis 3: Modular Specification and Verification of Smart Contracts . . . 111

Publications 113
Publications Linked to the Theses . . . 113
Additional Publications (Not Linked to Theses) . . . 114
Additional Work . . . 115

Bibliography 117

Introduction

> *"Computer science has become pervasive in production, transportation, infrastructure, health care, science, finance, administration, defense, and entertainment. Programs are the most complex machines built by humans, and have huge responsibilities for human safety, security, health, and well-being. These developments have exacerbated the challenges and, at the same time, dramatically increased the need for correct programs and, hence, for computer-aided verification."* [Cla+18]

As Clarke *et al.* states [Cla+18], our need for trust and reliance on correctly operating computer systems and programs is rapidly increasing. Such systems are often found in *critical* environments, where an error can lead to severe damage (e.g. industrial controllers) or financial consequences (e.g. asset management). While there is a wide variety of verification methods ranging from simple compiler checks to testing and runtime monitoring, *formal techniques* are also gaining traction in critical domains. Formal verification techniques have a sound mathematical basis and can both show the presence or prove the absence of certain kinds of errors. Rigorous reasoning about the operation of a computer system or program traces back several decades to seminal works of McCarty [McC62], Floyd [Flo67], Hoare [Hoa69] and Dijkstra [Dij76]. Despite early results proving that most of the interesting problems (e.g. termination) are theoretically undecidable [Tur36; Chu36; Ric53], there has been a great interest in developing approaches that can be effectively applied for practical cases.

Automated formal verification[2] [DKW08] gained a boost when *model checking* [Cla+18] was introduced, which examines whether a formal *model* (representation) of the system meets a formally specified *property* by analyzing all possible *states* and *transitions* (i.e. the *state space*) of the model. In this dissertation, we are addressing *discrete* systems, where the behavior of the system can be expressed in terms of discrete states and transitions. Early works explicitly enumerated the state space [CE82; QS82], which rarely scaled to programs and systems of a practical size. Nevertheless, the promise of formal correctness guarantees has spawned a great interest, and a wide variety of approaches have been developed. Symbolic methods [Bur+90] can encode the state space in a more compact way using decision diagrams [Bry86]. Partial order reduction techniques [Val91; God91; Pel93] exploit independence between transitions to traverse only a subset of the state space. Bounded model checking [Bie+99] exploits the advances in SAT solving (as demonstrated by competitions [Jär+12]) to find errors up to a given depth by encoding paths as formulas. Later, k-induction [SSS00] generalized it to be able to prove correctness as well (even for infinite state spaces [MRS03]). Abstraction-

[2] This work focuses on automated techniques, i.e. we are not considering semi-automated theorem provers [HUW14].

and abstraction-refinement-based techniques [CGL94; Cla+03] allow compact representation of (potentially infinite) state spaces using different abstract domains (e.g. predicates [GS97] or variable visibility [CGS04]). Modular specification and verification [Mül02] checks modules independently from others by only relying on their specification. IC3 [Bra11] iteratively strengthens an inductive invariant by relying on SAT solvers. However, despite the advances, there are open questions that have not yet been addressed to a full extent, and each new application or problem domain spawns new challenges. This dissertation targets such challenges in order to make verification more *effective*.

Properties and Challenges

From the theoretical point of view, two widely studied properties of formal verification are *soundness* and *completeness* [JM09; Bey12; Mey19] as illustrated in Figure 1. A system or program might behave desirably (with respect to a property) or might have violating behaviors. When formal verification is applied, it can either result in a pass (property holds) or a reject (property is violated).

Soundness. An analysis is called *sound* if it does not miss any violations to the property. Missed violations (also called false negatives) are critical because they lead to a misbelief of a correctly operating system.

Completeness. Analogously, an analysis is called *complete* if it only reports real violations of the property. In most cases, a reported violation (error) is accompanied by a trace leading to the error that can be reproduced in the original system. Therefore, false alarms (non-real errors) can be ruled out by simulating the reported trace on the original system. This usually requires manual effort, so an overwhelming amount of false alarms can make the approach less appealing in practice.

		Formal verification	
		Passes	Rejects
System or program	Desirable behavior	Accepted desirables	False alarms (incomplete)
	Violating behavior	Missed violations (unsound)	Caught violations

Figure 1: Illustration of the possible outcomes of verification compared to the real behavior [Mey19]. A sound analysis should not miss violations, while a complete one must not report false alarms.

In the current work, we are not considering soundness and completeness explicitly. The algorithms and background logics have a sound mathematical basis and have been used in various contexts; many of them also having formal proofs. Unsound and incomplete behavior is often introduced while translating the high-level model to the mathematical formalism, but translation validation is a research area on its own and is out of scope for this work. We raise our confidence in soundness and completeness by evaluating our approaches on various real-life examples and standard benchmarks.

Soundness and completeness are essential properties, but they only apply if verification terminates with a conclusive answer. In practical settings, usually, a broader set of properties and challenges

must be considered (Figure 2), such as the *expressive power* and *efficiency* of an approach and the set of problems on which it can terminate with a *conclusive answer*.

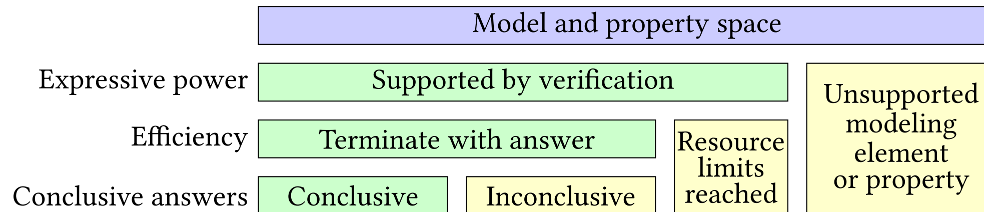


Figure 2: Possible outcomes of verification in practice. When verification cannot give a conclusive answer, it can terminate inconclusively, reach its resource limits or encounter an unsupported feature.

Expressive power. Engineers and programmers usually describe their systems and programs in some high-level modeling or programming language. High-level system models and properties are translated through a series of transformations to low-level mathematical formalisms (e.g. automata) and properties (e.g. temporal logic) on which verification algorithms operate. The *expressive power* of a verification approach is determined by the supported modeling formalism and property. Note that the expressive power of the low-level formalism and property also determines the set of high-level modeling elements and specification constructs that can be used. For example, to verify a protocol with unbounded communication channels, the algorithm should be able to handle infinite state spaces.

Efficiency. In practical applications, formal verification is limited by various resources such as CPU time or memory consumption. In the context of this dissertation, we consider a formal verification approach *efficient* if it allows scalable reasoning on systems of practical size and complexity. It is hard to define what "scalable" means explicitly because it also depends on the application domain. An interactive verifier built into an IDE should not take longer than a few seconds. Verification integrated into CI environments can run up to a few minutes [Cal+15; Cho+20]. Competitions [Bey17; Cab+16; Amp+19] usually allow larger execution times (15–60 minutes), and in some domains, it might also be acceptable to run analyses overnight for multiple hours.[3]

Conclusive answers. Algorithms might also encounter some undecidable case or unsupported subclass where they stop and report an *inconclusive answer*. This can happen, for example, when an approach over- or under-approximates the state space and can only prove or falsify the property but not both. A typical example is bounded model checking [Bie+99], which terminates with an inconclusive result if the bound is reached without finding a violation. Terminating with an inconclusive result is better than exhausting resources or reporting a wrong answer, but ideally, the number of such cases should also be minimized.

Trade-offs. It is hard (or sometimes even theoretically impossible) to achieve all the above properties to a full extent in a general setting [JM09]. For example, lifting the expressive power of the algorithm might make the problem theoretically undecidable, and thus the algorithm cannot be conclusive for all cases. Also, efficient reasoning often involves abstractions, which can introduce falsely reported errors, i.e. incompleteness.

[3] Based on personal communication with Dániel Darvas, the developer of a PLC verification tool [DFB15] at CERN.

Challenges. In this dissertation, we focus on the following three challenges.

1. *Expressive power*: How can we support expressing and checking high-level modeling formalisms and functional properties?
2. *Efficiency*: How can we increase the efficiency of an approach to be able to terminate for a broader set of system models and programs of practical size?
3. *Conclusive answers*: How can we increase the set of problems where verification terminates with a conclusive answer?

> **Objective.** The objective of the dissertation is to achieve a trade-off that is *effective* in practice by balancing the focus between the challenges in the different problem domains.

Problem Domains and Contributions

In this dissertation, we target effective verification in three different problem domains using different modeling formalisms and verification approaches:

1. concurrent and asynchronous systems (Thesis 1),
2. embedded software code (Thesis 2) and
3. blockchain-based decentralized systems (Thesis 3).

An overview of the contributions can be seen in Figure 3. Systems and programs in each domain are usually designed or written in some higher level language that is suitable for engineers and developers. This representation is first translated into a formal model and a property. A verification algorithm then checks whether the model satisfies the property by systematically exploring its behavior. During this process, the algorithm translates the validity of the property into formulas and equations, called *verification conditions* (VCs), and relies on some background logic to solve them. A filled background highlights my own contributions, with the corresponding subtheses numbered in ellipses (also referenced later in the text). As discussed previously, each domain puts more emphasis on different challenges. These challenges – namely expressive power, efficiency, and conclusive answers – are summarized in Figure 4.

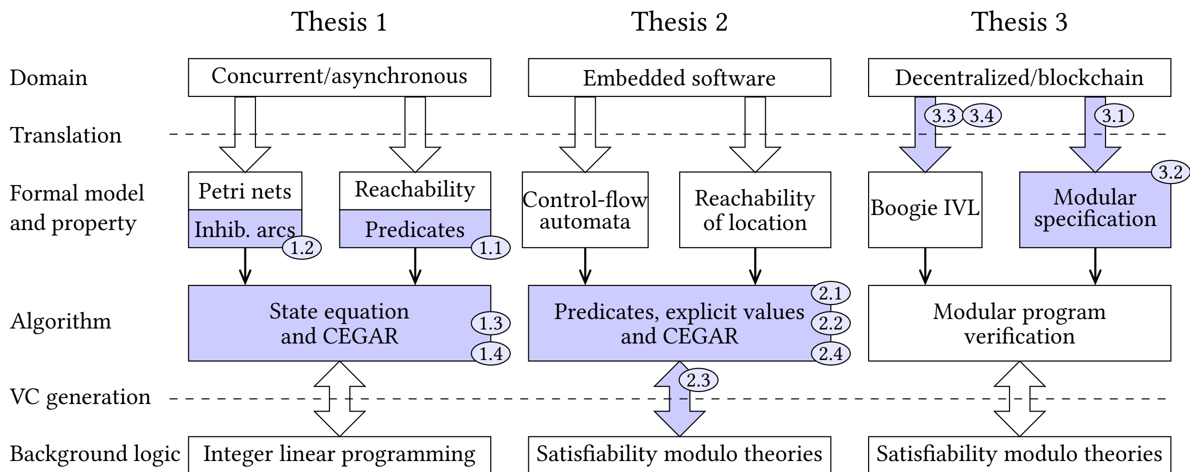


Figure 3: Overview of the problem domains and the verification approaches used in each thesis. Own contributions are denoted with a filled background.

	Thesis 1	Thesis 2	Thesis 3
Expressive power	(1.1) (1.2)		(3.1) (3.2)
Efficiency		(2.1) (2.2) (2.3) (2.4)	(3.3) (3.4)
Conclusive answers	(1.3) (1.4)		

Figure 4: Overview of the challenges addressed by each thesis.

Concurrent and Asynchronous Systems

Concurrent systems consist of multiple components interacting together, often in an asynchronous way, to achieve some common goal. Some examples include mutual exclusion protocols, scheduling processes, and manufacturing systems. The main focus of verification, in this case, is usually on the communication, the interactions, and the protocols between the participants. However, due to the high number of possible interleavings between the individual executions, the state space of these systems can often grow at an exponential (or even higher) rate with the number of participants. Furthermore, unbounded protocols can even yield an infinite state space.

Petri nets [Mur89] offer a compact representation, providing both structural and dynamical analysis. A Petri net is a directed bipartite graph with *places* and *transitions*. Places are *marked* with a number of *tokens*, describing the current state of the modeled system. Transitions change the distribution of tokens (i.e. the marking) by removing and producing tokens in connected places. Many interesting properties can be formulated by the so-called *reachability problem* [Mur89], i.e. deciding if a given state (marking) is reachable from the initial state of the net. Reachability is decidable [May81; Kos82], but has at least a non-elementary complexity [Cze+19].

There has been an extensive body of work on efficient approaches for solving Petri net reachability [Amp+19]. One appealing algorithm [WW11] uses the *state equation* of Petri nets to over-approximate the reachability problem. The state equation is a structural analysis technique based on *integer linear programming* (ILP) [Sch86]. A notable feature of the state equation is that – as a structural technique – it is independent of the size of the state space. Thus, it is capable of handling very large or even infinite state spaces efficiently. However, the feasibility of the state equation is only a necessary, but not a sufficient condition for reachability. Therefore, if there is no solution to the state equation, the target state is not reachable. Otherwise, the solution must be checked (simulated) in the Petri net for feasibility. In the case of an infeasible solution, the state equation is extended with additional constraints to become a more precise over-approximation and to obtain a different solution. The process is repeated until the state equation becomes infeasible, or a feasible solution is found. This can also be seen as an application of the so-called *counterexample-guided abstraction refinement* (CEGAR) approach [Cla+03] to Petri nets.

> **Thesis 1 objectives.** While the algorithm has proven its efficiency at the Model Checking Contest [Kor+12], its expressive power was limited to basic Petri net reachability, and no discussion was available on the problems on which it gives a conclusive answer. The main objectives of this research are to examine the problems on which the algorithm gives a *conclusive answer* and to lift its *expressive power* to extended Petri nets and more general properties.

Thesis 1: Extensions to the CEGAR Approach on Petri Nets

The authors only published a partial proof on the soundness of their algorithm and did not examine the set of problems on which it gives a conclusive answer [WW11]. In our initial work, we proved that one of the heuristics in their algorithm is unsound, i.e. a reachable state might be determined as unreachable [c4], and we also suggested a fix [j1]. We also showed a whole subclass of Petri nets for which their algorithm terminated with an inconclusive answer [c4]. In this thesis, we define the concept of *distant invariants* and propose a *new iteration strategy* (1.3), which extends the class of reachability problems that could be analyzed [c5]. Despite the extension, the improved algorithm can still give inconclusive answers, but we provide theoretical investigations on its limitations [c5].

Another limitation of the original algorithm is that it only works for Petri nets without any extensions. One particularly interesting extension is the inhibitor arc construct, which allows testing the lack of tokens at a place, lifting the expressive power of Petri nets to be Turing complete [Pet81]. We extend the constraint generation heuristic of the original algorithm to be able to *handle inhibitor arcs* (1.2) [c4]. Although reachability with inhibitor arcs is undecidable in general [Chr99], we present examples where our extension works.

To further improve the expressive power of the analysis, we extend the original algorithm to be able to *handle reachability of predicates* (1.1) [c4]. In this generalized version of reachability, one can define an arbitrary linear condition (predicate) over the state to be reached. This improves the expressive power of the algorithm as, for example, it allows to specify the state to be reached partially (e.g. one component in a larger system).

Although the algorithm approximates the state space with equations, the solution space still has to be traversed. We experiment with breadth- and depth-first search strategies and propose a *hybrid search strategy* (1.4) (based on a new partial order between solutions) to combine their strengths [c5].

We implemented the original algorithm and its extensions in the PetriDotNet modeling and analysis tool [c7], which is freely available[4] and used in education and research projects at the Budapest University of Technology and Economics. We also evaluate the new contributions on roughly 40 input models (from the Model Checking Contest [Kor+12] and some custom models). Results show that the new algorithms could outperform existing tools and approaches on various inputs in terms of conclusive answers and expressive power [c5].

Embedded Software Code

Safety critical software usually operates in embedded systems or controllers. Such programs are often written in C or a similar lower level language with a restricted set of elements and constructs. Some examples include industrial controller codes and event-driven systems. A widely used formal representation for such programs is the *control-flow automaton* (CFA) [BHT07]. A CFA is a graph-based formalism where nodes correspond to program *locations* and *edges* capture control-flow with *operations* over the program variables. Many interesting properties can be formalized by checking if a distinguished *error location* can be reached in the CFA. Examples include failing assertions, indexing out-of-bounds, division by zero, and so on [Bey15].

However, a significant challenge in software model checking is the large state space implied by data variables with rich domains (e.g. integers and arrays). This issue is often addressed by *abstraction. Counterexample-guided abstraction refinement* (CEGAR) [Cla+03] is an automated verification approach that works by iteratively constructing and refining abstractions for the system. Many variants of CEGAR have been developed over the years as different strategies are more suitable for different

[4] http://petridotnet.inf.mit.bme.hu/en/

kinds of programs. A generic CEGAR approach consists of two main parts [j3]. First, the *abstraction* phase builds an *abstract reachability graph* (ARG) using an initial (usually coarse) precision. The ARG represents the abstract state space under some abstract domain, such as *explicit values* [BL13] or *predicates* [GS97]. Explicit values only track a subset of the system variables, whereas predicates keep track of different facts and relationships between the variables using logical formulas. The ARG is an over-approximation of the original state space, therefore if the error location cannot be reached, the original system is also correct. Otherwise, an abstract counterexample (a trace leading to the error location) exists. The *refinement* phase starts by checking the feasibility of this counterexample in the original system. If it is feasible, the system is incorrect. Otherwise, the precision of the abstraction is refined by inferring new variables or facts to be tracked [j3], and the ARG is pruned to exclude the spurious counterexample. In the next iteration, abstraction can continue with the refined precision, and these steps are repeated until the error location can be proved to be unreachable or a feasible counterexample is found. The CEGAR algorithm relies on *satisfiability modulo theories* (SMT) [BT18; BHM09] in the background to build the ARG and to refine the precision.

> **Thesis 2 objectives.** Despite applying abstraction and CEGAR, scalability is still a major limiting factor in software model checking. Successful verification usually requires the combination of multiple approaches [BLW15a; BDW15; JD16] or a portfolio of different methods [Tul+14; Dem+17; Dar+18; GD19; RW19]. The main objective of this research is to improve the *efficiency* of the state of the art by developing new strategies for both abstraction and refinement by novel extensions and combinations of existing approaches.

Thesis 2: Efficient Strategies for CEGAR-based Software Model Checking

In our prior work, we defined a generic CEGAR framework for programs described by transition systems to be able to combine different approaches [c6]. This framework successfully facilitated the use of predicates and explicit values and incorporated different interpolation strategies. Later, we generalized this framework to also support programs described by control-flow automata [c9].

This leads us to this thesis, where we develop various improvements to both the abstraction and the refinement phases of CEGAR [j3]. For abstraction, we define an extension for the explicit-value domain that can perform a *limited enumeration* (2.1) of possible successor states when an expression cannot be precisely evaluated (due to the nature of abstraction). While this has a minimal performance penalty, it can be compensated later by the increased precision. We also propose a new *search strategy* (2.2) in the abstract state space that uses structural information from the program about the error location to guide the search more efficiently towards counterexamples. This approach can also help when checking correct programs because CEGAR encounters (abstract) counterexamples during intermediate steps.

For refinement, we develop a *backward search-based interpolation* strategy (2.3) to track the reason of infeasibility of abstract counterexamples back to the earliest point in the program. We also introduce an approach that collects *multiple counterexamples* (2.4) during abstraction and refines them at once, allowing information to be exchanged between the different counterexamples. Both contributions aim to yield a faster convergence to the appropriate precision.

We implemented the CEGAR algorithm and its improvements in the open-source[5] Theta verification framework [c9]. We also evaluate the new contributions on 445 input models from the Competition on Software Verification [Bey15] and 90 input PLC programs from CERN [Fer+15]. Results highlight various categories of inputs where the new contributions improved efficiency remarkably.

[5] https://github.com/FTSRG/theta

Blockchain-Based Decentralized Systems

Blockchain-based distributed ledgers are aiming to replace centralized solutions that require a trusted intermediary (e.g. banks). Early applications of the blockchain, such as the Bitcoin [Nak08], focused on implementing cryptocurrencies, i.e. digital money. Their success generated enormous attention, and later more general solutions emerged, for example, Ethereum [Woo17]. In the general setting, the ledger allows the deployment of programs (so-called *smart contracts* [Sza94]) that can store an arbitrary state (as data) on the blockchain and enable manipulating their data via transactions [AW18]. However, high-profile bugs and vulnerabilities highlighted that smart contracts are often prone to critical errors [ABC17; DMH17; Dat18]. Although the code of the contracts is usually small, it often carries a significant amount of value per line (e.g. by managing assets or tokens) [OHJ20].

While there have been various works on verifying smart contracts with static analysis [Tsa+18; Luu+16; Mue18; FGG19] and theorem proving [Hil+18; Hir17], not much effort had been put into the automated verification of high-level, *functional properties* of contracts. Due to the transactional behavior of the blockchain, *modular specification and verification* [Mül02] is an appealing approach for checking smart contracts. *Boogie* [DL05] is an intermediate verification language (IVL), which is supported by different backends, including a modular verification engine [Bar+06]. The units of verification in Boogie are the procedures, which can be annotated with specification expressions such as pre- and postconditions. Modular program verification checks if the specification of each procedure is satisfied by assuming the related modules' specifications to hold. This is achieved by encoding each procedure as SMT formulas (verification conditions) and discharging them with SMT solvers.

> **Thesis 3 objectives.** The main objective of this research is to develop an *expressive* and *efficient* modular specification and verification approach for checking high-level functional properties of smart contracts by translating them to the Boogie IVL.

Thesis 3: Modular Specification and Verification of Smart Contracts

In this thesis we define a modular specification and verification approach for smart contracts written in the Solidity language [Eth18]. We adapt various existing *specification constructs* (3.1) (such as assertions, pre- and postconditions and invariants) to the context of smart contracts [c10]. Such properties can be specified in the code itself using *annotations* that extend the Solidity language. We also propose some *domain specific properties* (3.2) (e.g. sums of balances) that are not expressible directly in Solidity or the verification logic [c10].

We define a *translation* (3.3) from annotated Solidity contracts to the Boogie IVL [c10]. This allows us to discharge the verification conditions automatically by leveraging modular verification and SMT solvers. While a significant part of the translation is similar to standard program verification, there are various challenging blockchain-specific details that are not common in general programming languages. We develop an *encoding of arithmetic* (3.4) using modulo operations that captures the bit-precise semantics of execution, while also being scalable to practical bit-widths (256 bits) even with nonlinear arithmetic expressions [c10]. This opens up the possibility to check for integer under- and overflows without introducing an overwhelming amount of false alarms.

We implemented the translation in the open-source[6] tool SOLC-VERIFY [c10] based on the Solidity compiler and the BOOGIE verifier. We evaluate our approach on several annotated and unannotated real-life examples by finding bugs, fixing them, and proving correctness with minimal user effort.

[6] https://github.com/SRI-CSL/solidity

Chapter **1**

Extensions to the CEGAR Approach on Petri Nets

This chapter presents our extensions to the CEGAR approach on the reachability analysis of Petri nets. We start by introducing the basis of our work: Petri nets, the reachability problem, and the CEGAR algorithm (Section 1.1). Then, we propose our contributions: handling reachability of predicates and inhibitor arcs, extending the set of decidable problems with so-called distant invariants, and our hybrid search strategy (Section 1.2). We briefly discuss the implementation of the algorithm in PetriDotNet (Section 1.3) and perform an experimental evaluation (Section 1.4). Then, we put our work in context with related literature (Section 1.5). Finally, we summarize the thesis, highlight the contributions, and suggest future directions (Section 1.6).

1.1 Background

In this section, we introduce the theoretical background of our work. First, we present Petri nets (Section 1.1.1) and their reachability problem (Section 1.1.2). Then we introduce linear programming (Section 1.1.3) and describe a CEGAR-based algorithm for reachability analysis (Section 1.1.4).

1.1.1 Petri Nets

Petri nets [Mur89] – invented by C. A. Petri [Pet62] – are graph-based models for concurrent and asynchronous systems, providing both structural and dynamical analysis techniques.

> **Definition 1.1 (Petri net).** A discrete Petri net is a tuple $PN = (P, T, E, W)$, where
> - $P = \{p_0, p_1, \ldots, p_k\}$ is the finite set of *places*,
> - $T = \{t_0, t_1, \ldots, t_n\}$ is the finite set of *transitions*,
> - $E \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs* and
> - $W : E \mapsto \mathbb{Z}^+$ is the weight function assigning weights $w^-(p_j, t_i)$ to the arc $(p_j, t_i) \in E$ and $w^+(p_j, t_i)$ to the arc $(t_i, p_j) \in E$.

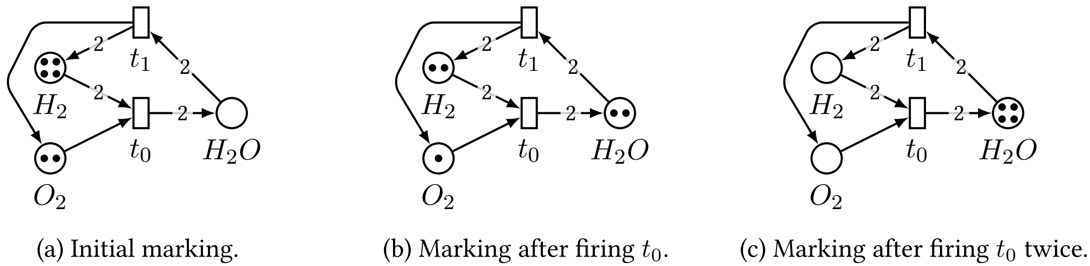
The state of the Petri net is described by the *marking*, which is a mapping $m : P \mapsto \mathbb{N}$. A place p is said to contain k *tokens* in a marking m if $m(p) = k$. The initial marking is denoted by m_0.

In the graphical representation of a Petri net, places are denoted by circles, transitions by rectangles, and arcs by arrows. If the weight of an arc is one, it is usually not labeled. The token distribution is denoted by numbers or dots inside the places. An example can be seen in Figure 1.1.

Dynamic behavior. A transition $t \in T$ is *enabled* in a marking m, if $m(p_j) \geq w^-(p_j, t)$ holds for each $p_j \in P$ with $(p_j, t) \in E$. An enabled transition t can *fire*, consuming $w^-(p_j, t)$ tokens from places $p_j \in P$ with $(p_j, t) \in E$ and producing $w^+(p_j, t)$ tokens in places $p_j \in P$ with $(t, p_j) \in E$. The firing of a transition t from a marking m is denoted by $m[t\rangle m'$ where m' is the marking obtained after firing t. Formally, $m'(p) = m(p) + w^+(p, t) - w^-(p, t)$ for each $p \in P$ (where w^+ and w^- is assumed to be 0 if there is no arc).

A word $\sigma = t_1 t_2 \dots t_n \in T^*$ is a *firing sequence*. A firing sequence is *realizable* in a marking m and leads to m' (denoted by $m[\sigma\rangle m'$), if $m[t_1\rangle \dots [t_n\rangle m'$. The *Parikh image* of a firing sequence σ is a vector $\wp(\sigma) \colon T \mapsto \mathbb{N}$, where $\wp(\sigma)(t_i)$ is the number of the occurrences of t_i in σ. The empty firing sequence is denoted by ε.

Example. *A simple Petri net modeling a chemical process can be seen in Figure 1.1. The net has three places (H_2, O_2, H_2O) and two transitions (t_0, t_1). Figure 1.1a shows the initial marking, where only t_0 is enabled. If t_0 fires, the marking seen in Figure 1.1b is reached, where both t_0 and t_1 are enabled. If now t_1 fires, the initial marking is reached. Otherwise, if t_0 fires again, the marking in Figure 1.1c is reached where now only t_1 is enabled.*

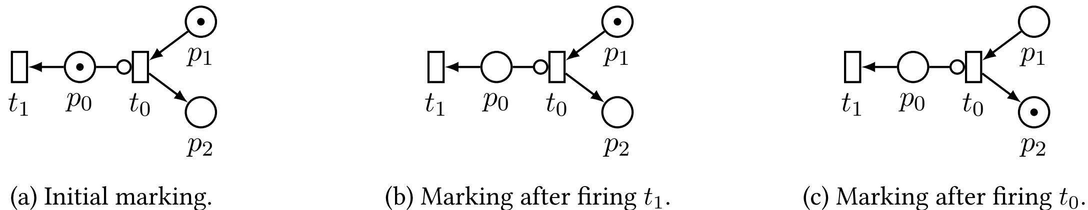


(a) Initial marking. (b) Marking after firing t_0. (c) Marking after firing t_0 twice.

Figure 1.1: Example Petri net modeling a chemical process (based on an example in [Mur89]).

Inhibitor arcs. Petri nets can be extended with *inhibitor arcs* to become a tuple $PN_I = (PN, I)$, where $I \subseteq (P \times T)$ is the set of inhibitor arcs. There is an extra condition for a transition $t_i \in T$ with inhibitor arcs to be enabled: for each $p_j \in P$, if $(p_j, t_i) \in I$, then $m(p_j) = 0$ must hold. With the ability to test for emptiness, Petri nets extended with inhibitor arcs are *Turing complete* [Pet81], which also puts a limitation on available analysis methods [Bus02].

Example. *An example net containing inhibitor arcs can be seen in Figure 1.2. Figure 1.2a shows the initial marking, where t_0 is disabled by the inhibitor arc connecting p_0 to t_0. After firing t_1, p_0 has zero tokens (Figure 1.2b), so t_0 can now fire. The marking in Figure 1.2c is reached after firing t_0.*



(a) Initial marking. (b) Marking after firing t_1. (c) Marking after firing t_0.

Figure 1.2: Example net illustrating transition firings with inhibitor arcs.

1.1.2 Reachability Problem

A marking m' is *reachable* from m if a realizable firing sequence $\sigma \in T^*$ exists for which $m[\sigma\rangle m'$ holds. The set of all reachable markings from the initial marking m_0 of a Petri net PN is denoted by $R(PN, m_0)$.

> **Definition 1.2 (Reachability problem).** The reachability problem of Petri nets is to decide if $m' \in R(PN, m_0)$ holds for a given marking m'.

Reachability analysis aims to solve the reachability problem by finding a realizable firing sequence $m_0[\sigma\rangle m'$. The reachability problem is *decidable* [May81; Kos82], but its complexity is not precisely known yet. It was proven to be at least EXPSPACE-hard [Lip76] in 1976, and this lower bound was just recently lifted to be *non-elementary* [Cze+19]. An upper bound was first found in 2015 [LS15] and was recently improved to be *Ackermannian* [LS19]. Using inhibitor arcs, the reachability problem, in general, is *undecidable* [Chr99].

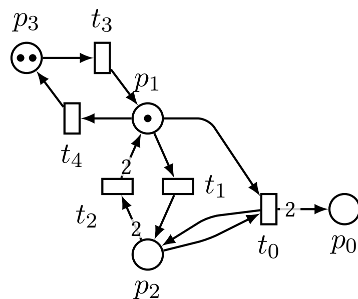
The reachability problem can be generalized to predicates in the following way. We define a *predicate* as a linear inequality on markings of the form $Am \geq b$, where A is a matrix and b is a vector of coefficients [EM00].

> **Definition 1.3 (Reachability problem of predicates).** The reachability problem of predicates for Petri nets is to decide if a reachable marking $m' \in R(PN, m_0)$ exists, for which a given predicate $Am' \geq b$ holds.

1.1.2.1 State Equation

The *incidence matrix* of a Petri net is a matrix $C_{|P|\times|T|}$, where $C(i,j) = w^+(p_i, t_j) - w^-(p_i, t_j)$. The element $C(i,j)$ represents the change in the number of tokens in p_i after firing t_j.

Example. *The incidence matrix of the Petri net in Figure 1.3a can be seen in Figure 1.3b. Note that the Petri net cannot always be restored from the incidence matrix, e.g. the two arcs between t_0 and p_2 appear as a zero in the matrix.*



(a) Petri net.

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ -1 & -1 & 2 & 1 & -1 \\ 0 & 1 & -2 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

(b) Incidence matrix.

Figure 1.3: Example net and its incidence matrix. Rows and columns correspond to places and transitions respectively. Each cell denotes the change in the number of tokens at a given place if a given transition fires.

A marking m of a Petri net can be written as a column vector (of size $|P|$), where the ith element is the token count of place $p_i \in P$. The *firing vector* u_j of a transition $t_j \in T$ is a column vector (of size

$|T|$) filled with zeros, except the jth place, which is one. Due to the definition of the incidence matrix, if t_j is enabled under m, then the marking m' after firing t_j can be obtained by $m' = m + Cu_j$. This can be generalized for a firing sequence $\sigma \in T^n$ by $m' = m + Cu_{j_1} + \ldots + Cu_{j_n}$ where $u_{j_1}, \ldots u_{j_n}$ are the firing vectors of transitions in σ. Since matrix-vector multiplication is distributive, substituting $x = u_{j_1} + \ldots + u_{j_n}$ into the previous equation yields the *state equation.*

> **Definition 1.4 (State equation).** Given a Petri net with is incidence matrix C and an initial and target marking m and m', the state equation has the form $m + Cx = m'$.

Any vector $x \in \mathbb{N}^{|T|}$ fulfilling the state equation is called a *solution.* Note that for any realizable firing sequence σ leading from m to m', the Parikh image of the firing sequence fulfills the equation $m + C\wp(\sigma) = m'$. On the other hand, not all solutions of the state equation are Parikh images of a realizable firing sequence. Therefore, the existence of a solution for the state equation is a necessary but not sufficient criterion for reachability. A solution x is called *realizable* if a realizable firing sequence σ exists with $\wp(\sigma) = x$.

Example. *Consider the Petri net with its incidence matrix in Figure 1.3, and markings $m = (0, 1, 0, 2)$ and $m' = (2, 0, 0, 2)$. For example, $x_1 = (1, 0, 0, 0, 0)$ is a solution to the state equation, but it is not realizable. On the other hand, $x_2 = (1, 2, 1, 2, 2)$ is also a solution and it is realizable by the firing sequence $t_3t_3t_1t_1t_0t_2t_4t_4$.*

1.1.2.2 T-Invariants

A vector $y \in \mathbb{N}^{|T|}$ is called a *T-invariant* if $Cy = 0$ holds. A realizable T-invariant represents the possibility of a cyclic behavior in the modeled system since its complete occurrence does not change the marking. However, while firing the transitions of the T-invariant, some intermediate markings can be of interest. If each component of the T-invariant y is either zero or one we also denote y by enumerating the components with value one, e.g. $y = (1, 0, 1, 0)$ can be denoted by $y = \{t_0, t_2\}$. Note that any linear combination of T-invariants is also a T-invariant.

Example. *Consider the Petri net with its incidence matrix in Figure 1.3. For example, $y_1 = (0, 2, 1, 0, 0)$ and $y_2 = (0, 0, 0, 1, 1)$ are T-invariants. Moreover, their linear combinations, e.g. $y_1 + 2y_2 = (0, 2, 1, 2, 2)$, are also T-invariants.*

1.1.2.3 Solution Space

The solution space of the state equation $m + Cx = m'$ is semi-linear. Each solution x can be written as the sum of a *base solution* and the linear combination of T-invariants [WW11], which can formally be written as $x = b + \sum_i n_i y_i$, where $b \in \mathbb{N}^{|T|}$ is the base solution and $n_i \in \mathbb{N}$ is the coefficient of the T-invariant $y_i \in \mathbb{N}^{|T|}$.

Example. *Consider the Petri net in Figure 1.3. The (realizable) solution $x = (1, 2, 1, 2, 2)$ can be written as $x = b + y_1 + 2y_2$, where $b = (1, 0, 0, 0, 0)$ is a base vector and $y_1 = (0, 2, 1, 0, 0)$ and $y_2 = (0, 0, 0, 1, 1)$ are T-invariants.*

1.1.3 Linear Programming

Linear programming (LP) is a mathematical approach for finding an optimal solution for a set of linear inequalities and a linear objective function [Sch86]. The canonical form of a linear programming problem is the following:

$$\begin{aligned} \text{minimize} \quad & c^T x, \\ \text{subject to} \quad & Ax \leq b \text{ and} \\ & x \geq 0, \end{aligned}$$

where x is the vector of variables, b, c are vectors, and A is a matrix of coefficients. The feasible region of the linear programming problem is a convex polyhedron, which is defined by the intersection of a finite number of half-spaces. Linear programming aims to determine the point in the feasible region, where the objective function is minimal (or maximal) if such a point exists. Linear programming can be solved in polynomial time[1] [Sch86].

Integer linear programming. When all the variables of x must be integers, the problem is called the *integer linear programming* (ILP) problem. Despite linear programming being polynomially solvable, integer linear programming is an NP-hard problem [Sch86].

While the definition of (I)LP assumes constraints to be inequalities of the form $Ax \leq b$, most solvers (such as lpSolve) support additional operators, including "$\geq$", "$=$", "$<$" and "$>$" (and their combination). To simplify our presentation in the rest of the thesis, we will assume the existence of such operators and that they can be reduced to (I)LP. For example, $Ax \geq b$ becomes $(-A)x \leq -b$, $Ax = b$ becomes $Ax \leq b$ and $Ax \geq b$, and $Ax < b$ becomes $Ax \leq b - 1$ (for ILP only).

The basis of our current work is an algorithm (Section 1.1.4) that extends the state equation with additional constraints to analyze reachability. This is an integer linear programming problem since the firing counts of transitions must be integers.

1.1.4 CEGAR for Petri Nets

In this section, we present an algorithm published by Wimmel and Wolf [WW11], which applies counterexample-guided abstraction refinement (CEGAR) to the state equation in order to efficiently solve the reachability problem of Petri nets.

1.1.4.1 Overview

Existential abstraction [CGL94] is a technique to over-approximate the state space by a simpler, abstract representation. Over-approximation means that for each concrete trace in the state space, there is a corresponding abstract trace. Therefore, if a property holds for all reachable abstract states, it also holds for each reachable concrete state. However, if there is a counterexample for which the property does not hold, it might be caused by the over-approximation. Thus, every counterexample must be examined whether it has a corresponding concrete counterexample in the original model. If a concrete counterexample exists, it is a proof that the property does not hold in the original model. Otherwise, the abstract counterexample is spurious, and the abstraction has to be refined using the information from the examination. This technique is called *counterexample-guided abstraction refinement* (CEGAR) and it is widely used in model checking [Bey+07; Cla+03; Joh+13].

Wimmel and Wolf published an algorithm [WW11], which applies the CEGAR approach to the reachability analysis of Petri nets by using the state equation as an over-approximating abstraction. Figure 1.4 shows an overview and each step is detailed in the following subsections.

[1] While it is polynomially solvable in theory, for practical cases algorithms with exponential worst-case complexity (e.g. the simplex algorithm) often perform better.

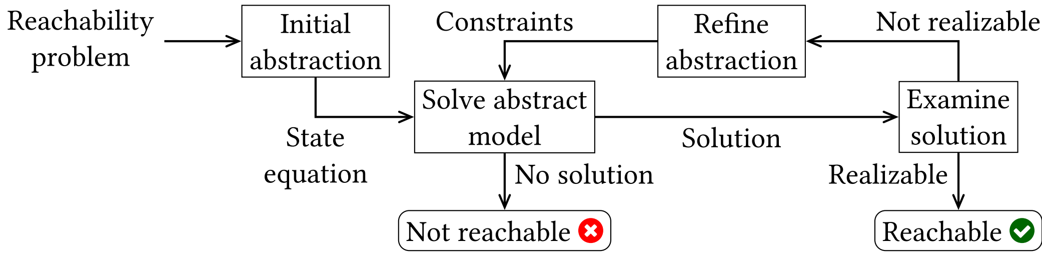


Figure 1.4: Overview of the steps of the Petri net CEGAR algorithm.

1.1.4.2 Initial Abstraction

The input of the algorithm is a reachability problem $m' \in R(PN, m_0)$, which is first transformed into the initial abstraction, namely the state equation of the form $m_0 + Cx = m'$. The state equation is a suitable abstraction because it is an over-approximation (due to being a necessary condition for reachability). Since the state equation is a structural analysis technique, it can potentially handle very large (or even infinite) state spaces.

1.1.4.3 Solving the Abstract Model

Solving the abstract model (i.e. the state equation) is an integer linear programming problem [Sch86]. The ILP solver yields a minimal solution with respect to the cost function. In the algorithm of Wimmel and Wolf [WW11], the sum of the firing count of transitions (each weighing one) is minimized in order to obtain firing sequences with the shortest length.[2]

The state equation is an over-approximation of the set of reachable markings, since its feasibility is a necessary, but not sufficient condition for reachability. Therefore, if no abstract solution exists, the target marking cannot be reached in the Petri net either. However, a solution of the abstract model may or may not be realizable by a firing sequence. Thus, further examination is needed.

1.1.4.4 Examining the Solution

The solution of the state equation is a vector $x \in \mathbb{N}^{|T|}$, where $x(t)$ denotes the number of times a transition $t \in T$ has to fire in order to reach m' from m_0. However, x does not include any information about the order of the transition firings and whether they are enabled. Thus, the algorithm has to explore the state space of the Petri net with a bound that each transition t can fire at most $x(t)$ times. A method for this traversal is discussed later in Section 1.1.4.6. If the target marking m' can be reached with this limit (i.e. x is realizable), it is a sufficient proof for reachability. Otherwise, x is a spurious solution, and the abstraction has to be refined.

1.1.4.5 Abstraction Refinement

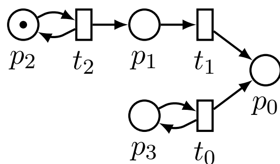
If a solution x is not realizable, the ILP solver has to be forced to generate a different solution. This can be done by adding additional *constraints* (i.e. linear inequalities over transitions) to the state equation. The following two types of constraints were defined by Wimmel and Wolf [WW11].

[2]We also experimented with arbitrary cost functions and a cost-based solution space traversal strategy to solve optimization problems [c14].

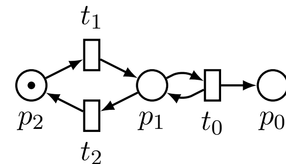
- *Jump constraints* have the form $|t_i| < n$, where $n \in \mathbb{N}$, $t_i \in T$ and $|t_i|$ represents the firing count of the transition t_i. Jump constraints can be used to obtain different base solutions, exploiting their pairwise incomparability.
- *Increment constraints* have the form $\sum_{t_i \in T} n_i|t_i| \geq n$, where $n_i \in \mathbb{Z}$, $n \in \mathbb{N}$, and $t_i \in T$. Increment constraints can be used to reach non-base solutions, i.e. some linear combination of T-invariants is added.

Example. *Consider the Petri net PN in Figure 1.5a with the reachability problem $(1,0,1,0) \in R(PN,(0,0,1,0))$. There are two base solutions for this problem: $(1,0,0)$ (firing t_0) and $(0,1,1)$ (firing t_1 and t_2). Since the ILP solver minimizes the firing count of transitions, it yields the solution $(1,0,0)$ first, which is not realizable. Using a jump constraint $|t_0| < 1$, the ILP solver can be forced to produce a different base solution $(0,1,1)$, which is realizable by t_2t_1.*

Example. *Consider the Petri net PN in Figure 1.5b with the reachability problem $(1,0,1) \in R(PN,(0,0,1))$. The only base vector for this problem is the vector $(1,0,0)$ (firing t_0), which is not realizable. Using an increment constraint $|t_1| \geq 1$ (which is $0|t_0| + 1|t_1| + 0|t_2| \geq 1$ in its full form), the ILP solver can be forced to add the T-invariant $\{t_1,t_2\}$ to the new solution $(1,1,1)$, which is realizable by $t_1t_0t_2$.*



(a) Example where a jump constraint is needed to produce a token in p_0.



(b) Example where an increment constraint is needed to produce a token in p_0.

Figure 1.5: Example nets illustrating jump and increment constraints.

After adding the new constraint, the state equation either becomes infeasible, or a new solution is obtained. Figure 1.6 presents the solution space. The bottom dots represent base solutions, while the cones represent the linear space formed by the T-invariants. The upper dots correspond to non-base solutions. Jumps are denoted by dashed arrows and increments by continuous arrows. The precise method for generating constraints and traversing the solution space is presented in later subsections, but first, *partial solutions* are introduced.

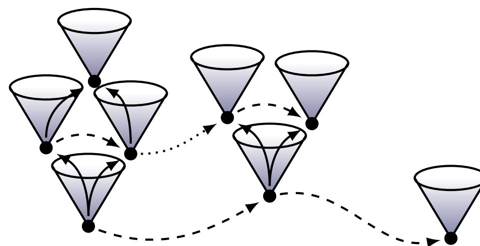


Figure 1.6: Illustration of the semi-linear solution space of the state equation [WW11]. Each cone represents a linear space over a solution. Dashed and solid arrows denote jump and increment constraints respectively.

1.1.4.6 Partial Solutions

Definition 1.5 (Partial solution). Given a Petri net $PN = (P, T, E, W)$ and a reachability problem $m' \in R(PN, m_0)$, a *partial solution* is a tuple $ps = (\mathcal{C}, x, \sigma, r)$, where:

- $\mathcal{C}$ is the set of (jump and increment) constraints, together with the state equation they define the ILP problem,
- x is the minimal solution (with respect to the cost function defined in Section 1.1.4.3) satisfying the state equation and the constraints belonging to the set $\mathcal{C}$,
- $\sigma \in T^*$ is a *maximal* realizable firing sequence, with $\wp(\sigma) \leq x$, i.e. each transition $t \in T$ can fire at most $x(t)$ times and enabled transitions must fire in some order,
- $r = x - \wp(\sigma)$ is the *remainder* vector.

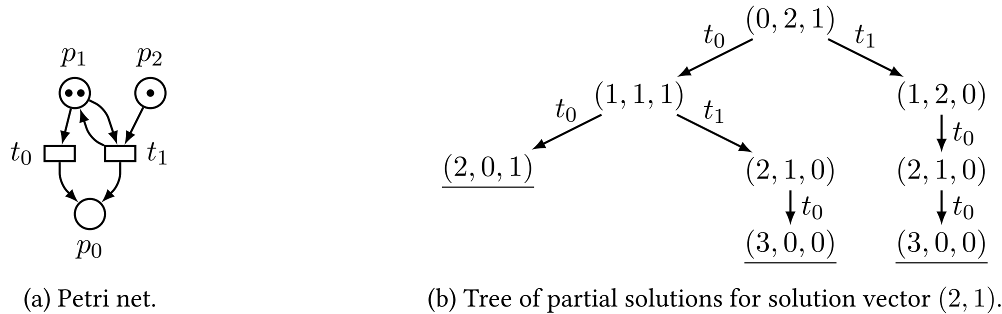
Partial solutions are generated while examining the solution x by exploring the state space of the Petri net. For this purpose, Wimmel and Wolf use a "brute force" method with some optimization [WW11] (e.g. stubborn sets [Sch99]). Conceptually, the algorithm builds a tree with markings as nodes and occurrences of transitions as edges. The root of the tree is the initial marking m_0, and there is an edge labeled by t between nodes m_1 and m_2 if $m_1[t\rangle m_2$ holds. On each path leading from the root of the tree to a leaf, each transition t_i can occur at most $x(t_i)$ times. Each path to a leaf represents a maximal firing sequence, thus a new partial solution. The marking reached is referred to as the *final marking* of the partial solution and is usually denoted by $\hat{m}$.

The basic algorithm (without any optimizations) can be seen in Algorithm 1.1. It maintains a queue starting with the initial marking m_0 and an empty firing sequence ε. While the queue is not empty, a node (m, σ) is removed (with respect to some search strategy and possibly some pruning optimizations [WW11]). Then, for each transition that is both enabled in m and can still fire based on the solution x, a successor is added. If there were no successors, the current node is a leaf and a corresponding partial solution is created.

Algorithm 1.1: Generate partial solutions for solution vector.

```
input  : (x, C): Solution vector and its constraints
         m_0: Initial marking
output: PSS: Partial solutions
1 PSS := ∅
2 queue := {(m_0, ε)} // Queue for pairs of markings and firing sequences
3 while queue ≠ ∅ do
4 |   m, σ := remove from queue
5 |   foreach enabled t_i under m with ℘(σ)(t_i) < x(t_i) do
6 |   |   m' := fire t from m, i.e. m[t⟩m'
7 |   |   queue := queue ∪ {(m', σt)}
8 |   if no nodes were added to queue for (m, σ) then PSS := PSS ∪ {(C, x, σ, x − ℘(σ))}
9 return PSS
```

Example. *Consider the Petri net in Figure 1.7a with the solution vector $x = (2, 1)$. The tree of partial solutions for x can be seen in Figure 1.7b. There are three different maximal firing sequences, thus three partial solutions. Note that there can be maximal firing sequences with different lengths. For example, $t_0 t_0$ has length 2, and $t_0 t_1 t_0$ has length 3. Both are maximal, because no transitions are enabled at their final marking.*



(a) Petri net.

(b) Tree of partial solutions for solution vector $(2, 1)$.

Figure 1.7: Example net and its tree of partial solutions for a given solution vector. Each leaf (underlined) is a maximal firing sequence, thus a partial solution.

A partial solution is called a *full solution* if $r = 0$ holds, i.e. $\wp(\sigma) = x$. Finding a full solution means that the solution vector x is realizable, and the algorithm can stop. Wimmel and Wolf proved that for each realizable solution of the state equation, a full solution exists. Furthermore, this full solution can be reached by continuously expanding the minimal solution of the state equation with constraints [WW11].

Consider now a partial solution $ps = (\mathcal{C}, x, \sigma, r)$, which is not a full solution. This means that some transitions could not fire enough times. There are three possible situations in this case:

1. x may be realizable by another firing sequence σ', thus a full solution $ps' = (\mathcal{C}, x, \sigma', 0)$ can be found in the tree of x.
2. By adding jump constraints, greater, but pairwise incomparable solutions can be obtained.
3. For transitions $t \in T$ with $r(t) > 0$ increment constraints can be added to increase the token count in the input places of t. The final marking m' must be unchanged, so this can be achieved by including additional T-invariants in the solution. During intermediate markings, these T-invariants can "borrow" tokens for transitions in the remainder vector.

1.1.4.7 Generating Constraints

When a partial solution is not a full solution, both jump and increment constraints can be added, but they are applied on a different level:

- Jump constraints are generated from solution vectors of the state equation.
- Increment constraints are generated from partial solutions (belonging to solution vectors).

Jump constraints. Given a solution vector x, for each transition $t_i \in T$ with $x(t_i) > 0$ a jump constraint c_i of the form $|t_i| < x(t_i)$ can be added to the state equation, as illustrated by Algorithm 1.2. If a new solution vector y_i is obtained after adding one of the constraints c_i, this process can be recursively repeated for y_i (see later in the main loop, presented in Section 1.1.4.9). Wimmel and Wolf proved that every base solution can be obtained using jump constraints [WW11].

Transforming jumps. Reaching non-base solutions requires increment constraints, but they might conflict previous jump constraints. As an example, consider that the solution $b_1 = (2, 0, 0)$ is reached with the jump constraint $|t_1| < 1$ from the minimal solution $b_0 = (0, 1, 0)$. If now we want to add the T-invariant $\{t_1, t_2\}$ to b_1 that would contradict with the previous constraint. Since jumps are only used to obtain pairwise incomparable solutions, they can be transformed into equivalent increment constraints using Algorithm 1.3.

Algorithm 1.2: Generate new solution vectors with jump constraints.

input : $(x, \mathcal{C})$: Solution vector and its constraints
output: XS: New solution vectors (and constraints) with jumps

```
1 XS := ∅
2 foreach t_i ∈ T with x(t_i) > 0 do
3 |   c_i := |t_i| < x(t_i)
4 |   if solution y_i exists for C ∪ {c_i} then XS := XS ∪ {(y_i, C ∪ {c_i})}
5 return XS
```

Algorithm 1.3: Transform jump constraints into increment constraints.

input : $\mathcal{C}$: Set of constraints
z: Minimal solution fulfilling $\mathcal{C}$
output: $\mathcal{C}'$: Set of constraints with no jumps

```
1 C' := each increment constraint of C
2 foreach t_i ∈ T do C' := C' ∪ {|t_i| ≥ z(t_i)}
3 return C'
```

Suppose that $\mathcal{C}$ is a set of jump and increment constraints, and z is the minimal solution fulfilling the state equation and $\mathcal{C}$. Let $\mathcal{C}'$ include each increment constraint of $\mathcal{C}$ and an additional increment constraint of the form $|t_i| \geq z(t_i)$ for each transition $t_i \in T$. Then, a vector $y \geq z$ is a solution of the state equation plus $\mathcal{C} \cap \mathcal{C}'$ if and only if y is a solution of the state equation and $\mathcal{C}'$. Furthermore, no solution smaller than z fulfills the state equation and $\mathcal{C}'$. This means that if we are interested in the solutions of the linear space over z, we can replace $\mathcal{C}$ with $\mathcal{C}'$, which no longer contains conflicting jump constraints [WW11].

Increment constraints. Let $ps = (\mathcal{C}, x, \sigma, r)$ be a partial solution with $r > 0$. This means that some transitions could not fire enough times. Wimmel and Wolf use a heuristic [WW11] to find the places and number of tokens needed to enable these transitions. If a set of places actually needs n ($n > 0$) tokens, the heuristic estimates a number from 1 to n. If the estimate is too low, this method can be applied again, converging to the actual number of required tokens. The heuristic – formalized by Algorithm 1.4 – consists of three steps: (1) building a dependency graph to determine places that need additional tokens, (2) determining the number of tokens needed, and (3) creating constraints based on the places and their token requirements.

Let $\hat{m}$ be the final marking of the partial solution ps, i.e. $m_0[\sigma\rangle\hat{m}$. The first step is to build a dependency graph $G = (P_0 \cup T_0, E)$, which consists of transitions that could not fire enough times (T_0) and places (P_0) that disable these transitions under $\hat{m}$. An edge (p, t) means that p disables t under $\hat{m}$, while an edge (t, p) means that firing t would increase the token count of p. Each source SCC[3] of the dependency graph has to be investigated, because it cannot get tokens from other components. For each source SCC the heuristic determines a tuple (P_i, T_i, X_i), where P_i is the set of places of the SCC, T_i is the set of transition of the SCC, and X_i is the set of transitions outside the SCC that depend on the current SCC.

The second step is to calculate the token requirement of each source SCC with the tuple (P_i, T_i, X_i). There are two possible cases.

[3]Source strongly connected component, i.e. one without incoming edges from other components.

- If $T_i \neq \emptyset$, then enabling one transition $t_i \in T_i$ may enable all the others, since t_i can produce additional tokens in some places of P_i. In this case n is the number of tokens required by the transition missing the least tokens.
- If $T_i = \emptyset$, then P_i can only consist of a single place p_i. This means that the token requirements of X_i must be fulfilled by p_i. However, transitions of X_i can also produce tokens in p_i, which has to be considered in the estimation. Transitions therefore, are ordered in groups: each group G_j consists of transitions $t \in X_i$ that produce j tokens in p_i. Firing the transitions of a group with the smallest value j last minimizes the leftover (denoted by c) at p_i. Transitions in the same group G_j can be processed at once, each using $w^-(p_i, t) - j$ tokens, except for the first one, which requires j additional tokens. The tokens produced by a group G_j can be consumed by the next group G_{j-1}. After processing each group, we get the estimated number n of tokens required to fire each transition in X_i.

Algorithm 1.4: Find increment constraints for a partial solution.

```
input  : ps = (C, x, σ, r): Partial solution with r > 0
         m_0: Initial marking
output: C': Constraints extended with new increment constraints
1  m̂ := final marking of ps, i.e. m_0[σ⟩m̂
2  // Step 1: build dependency graph G
3  T_0 := {t ∈ T | r(t) > 0}
4  P_0 := {p ∈ P | ∃t ∈ T_0 : w⁻(p,t) > m̂(p)}
5  E := {(p,t) ∈ P_0 × T_0 | w⁻(p,t) > m̂(p)} ∪ {(t,p) ∈ T_0 × P_0 | w⁺(p,t) > w⁻(p,t)}
6  C' := C
7  foreach source SCC_i in G = (P_0 ∪ T_0, E) do
8  |  P_i := SCC_i ∩ P_0
9  |  T_i := SCC_i ∩ T_0
10 |  X_i := {t ∈ T_0 \ SCC_i | ∃p ∈ P_i : (p,t) ∈ E}
11 |  // Step 2: estimate n
12 |  if T_i ≠ ∅ then  n := min_{t∈T_i}(Σ_{p∈P_i} max(0, w⁻(p,t) − m̂(p)))
13 |  else
14 |  |  p_i := single place of P_i // if T_i = ∅ then |P_i| = 1
15 |  |  sort X_i in groups G_j := {t ∈ X_i | w⁺(p_i,t) = j}
16 |  |  n := c := 0
17 |  |  foreach j with G_j ≠ ∅ downwards loop do
18 |  |  |  c := c + j + Σ_{t∈G_j} r(t) · (w⁻(p_i,t) − j)
19 |  |  |  if c > 0 then n := n + c
20 |  |  |  c := −j
21 |  // Step 3: create constraint c'
22 |  P' := P_i
23 |  T' := {t ∈ T | r(t) = 0 ∧ Σ_{p∈P'}(w⁺(p,t) − w⁻(p,t)) > 0}
24 |  c' := Σ_{t∈T'} Σ_{p∈P'}(w⁺(p,t) − w⁻(p,t)) · |t| ≥
   |     n + Σ_{t∈T'} Σ_{p∈P'}(w⁺(p,t) − w⁻(p,t)) · ℘(σ)(t)
25 |  C' := C' ∪ {c'}
26 return C'
```

The third step is to calculate an increment constraint. Let P' be the set of places, which require n additional tokens and T' be the transitions outside the remainder vector that can produce tokens in the places of P'. The left-hand side of the constraint c' consists of transitions weighted with the number of tokens produced in P'. The formula on the right-hand side is the estimate n plus the number of tokens already produced by the transitions in the firing sequence σ. This constraint will force transitions (with $r(t) = 0$) to produce tokens in the given places. Since the final marking has to be left unchanged, only a T-invariant can be added to the solution vector.

Finally, when applying the new constraints, three situations are possible depending on the T-invariants in the Petri net (see also later in the main loop, presented in Section 1.1.4.9):

- If the state equation and the set of constraints become infeasible, this partial solution cannot be extended to a full solution. Therefore it is no longer of interest.
- If the ILP solver can produce a solution $x' = x + y$ (with y being a T-invariant), new partial solutions can be found for x'.
  - If there is a new partial solution ps' where some transitions in the remainder vector could fire (compared to ps), this method can be repeated, eventually converging to a full solution.
  - If none of partial solutions helps to get closer to a full solution, repeating this method might lead the algorithm into an infinite loop: as the remainder r did not change, increment constraints will simply add the same invariant (y) again and again. A method to avoid this non-termination phenomenon will be discussed later in Section 1.1.4.8.

Reachability of a realizable solution. The following theorem of Wimmel and Wolf [WW11] states that if the reachability problem has a solution, it can be reached by the CEGAR approach: *If the reachability problem has a solution, a realizable solution of the state equation can be reached by continuously expanding the minimal solution with jump and increment constraints.*

1.1.4.8 Optimizations

Wimmel and Wolf also presented some methods for optimization [WW11]. In our current work, only the following, T-invariant filtering optimization is essential. After adding a T-invariant y to the partial solution $ps = (\mathcal{C}, x, \sigma, r)$, all the transitions of y may fire without enabling any transition in r, yielding a partial solution $ps' = (\mathcal{C}', x + y, \sigma', r)$ with $\wp(\sigma') = \wp(\sigma) + y$. The final marking and remainder vector of ps' is the same as in ps. Therefore the same T-invariant y would be added to the solution by the heuristic (Algorithm 1.4) again, possibly preventing termination. Thus, the optimization cuts the search space at ps'. This check is performed using Algorithm 1.5. Given the new partial solution ps', the optimization checks each partial solution ps^* belonging to each ancestor solution vector x of ps'. If the remainders are equal and the firing sequences differ in a T-invariant, then ps' can be skipped due to ps^*.

Algorithm 1.5: Check if a partial solution can be filtered based on T-invariants.

input : $ps' = (\mathcal{C}', x', \sigma', r')$: Partial solution to be checked
output: (skip or no skip, ps^*): Result and the other part. sol. that caused skipping (if exists)

1 **foreach** *solution vector* x *from ancestors of* ps' **do**
2   **foreach** *partial solution* $ps^* = (\mathcal{C}, x, \sigma, r)$ *of* x **do**
3   **if** $ps^* \neq ps'$ *and* $\wp(\sigma') - \wp(\sigma)$ *is T-invariant and* $r' = r$ **then return** *(skip,* ps^**)*
4 **return** *(no skip,* ps'*)*

Note that if a partial solution $ps' = (\mathcal{C}', x + y, \sigma', r)$ was skipped, while firing the transitions of y, the algorithm could get closer to enabling a transition in r (without reaching the limit where it becomes enabled). These "better" intermediate markings should be detected, and be used as new partial solutions (an example will be presented in Section 1.1.4.10). Wimmel and Wolf gave a definition for better intermediate markings, which we generalized in our former work [j1] as follows.

> **Definition 1.6 (Better intermediate marking).** An intermediate marking m_i is considered better than the final marking $\hat{m}$ of the firing sequence σ if there exists a transition t with $r(t) > 0$ and a place p with $(p, t) \in E$ for which $\hat{m}(p) < w^-(p, t) \ \wedge \ m_i(p) > \hat{m}(p)$ holds.

This means that t is disabled by p and p had more tokens in the intermediate marking m_i than in the final marking $\hat{m}$. In this case, the algorithm skips ps' but instead, continues from a new partial solution where the firing sequence σ was only fired up to the intermediate marking m_i.[4] Algorithm 1.6 presents a method to find better intermediate markings by replaying the firing sequence and checking the above definition at each step.

Algorithm 1.6: Get new partial solutions from better intermediate markings.

```
    input : ps = (C, x, σ, r): Skipped partial solution
            m_0: Initial marking
    output: PSS: New partial solutions
1  PSS := ∅
2  m̂ := final marking of ps, i.e. m_0[σ⟩m̂
3  foreach prefix σ_i of σ with length i (0 ≤ i ≤ |σ|) do
4  |  m_i := fire σ_i from m_0, i.e. m_0[σ_i⟩m_i
5  |  if ∃t ∈ T, ∃p ∈ P: r(t) > 0 ∧ (p,t) ∈ E ∧ m̂(p) < w⁻(p,t) ∧ m_i(p) > m̂(p) then
6  |  |  PSS := PSS ∪ {(C, x, σ_i, x − ℘(σ_i))}
7  return PSS
```

1.1.4.9 Main Loop of the Algorithm

Algorithm 1.7 summarizes the main loop of the CEGAR approach on Petri nets, building on the previously presented algorithm snippets (denoted by small caps). A dashed underline indicates new contributions to be presented later. For the baseline algorithm, their whole line should be ignored (removed).

The input of the algorithm is the Petri net PN with its initial marking m_0, and the initial set of constraints $\mathcal{C}_0$. For reachability, $\mathcal{C}_0$ is simply the state equation $m_0 + Cx = m'$, but in Section 1.2.1 we generalize it to allow an arbitrary linear predicate over the state to be reached. The algorithm starts by checking the initial conditions $\mathcal{C}_0$. If a solution x exists, it is added to the queue with an empty set of additional constraints. For simplicity of the presentation, we assume from this point on that the initial constraints $\mathcal{C}_0$ are always part of the ILP problem to be solved.

While the queue is not empty, an element is removed based on some search strategy (see Section 1.2.4 for different strategies). If the current element is a solution vector (with its constraints), we first use jump constrains to generate new solutions. Then, we build the tree of partial solutions and check if a full solution exists (terminating the algorithm). Later, we present an ordering and filtering over the partial solutions (Section 1.2.4.3), but for now we just put all of them in the queue.

[4] Note that the definition of partial solutions (Definition 1.5) requires the firing sequence to be maximal. We omit this restriction for these special cases when a partial solution is obtained via better intermediate markings.

Algorithm 1.7: Main loop of the Petri net CEGAR algorithm.

input : PN: Petri net
m_0: Initial marking
$\mathcal{C}_0$: Initial constraints
output: (Reachable, σ) or Not reachable or Inconclusive

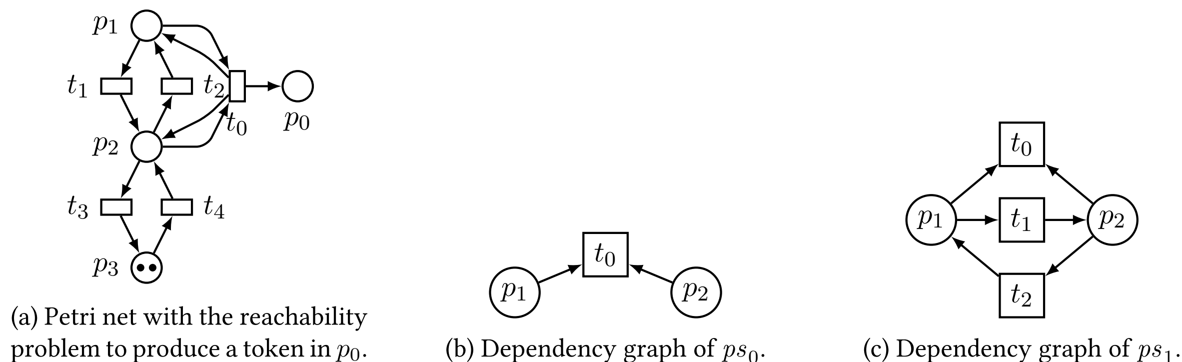
1 queue $:= \emptyset$ // Mixed queue of solution vectors (with constraints) and partial solutions
2 **if** *a solution* x *exists for* $\mathcal{C}_0$ **then** queue $:= \{(x, \emptyset)\}$
3 **while** queue $\neq \emptyset$ **do**
4 $\quad e :=$ remove element from queue // Based on some search strategy (see Section 1.2.4)
5 $\quad$ **if** e *is a solution vector* $(x, \mathcal{C})$ **then**
6 $\quad\quad$ queue $:=$ queue $\cup$ SolutionsWithJumps$(x, \mathcal{C})$ ➔ Algorithm 1.2
7 $\quad\quad$ $PSS :=$ PartialSolutions$(x, \mathcal{C}, m_0)$ ➔ Algorithm 1.1
8 $\quad\quad$ **if** $\exists(\mathcal{C}, x, \sigma, r) \in PSS$ *with* $r = 0$ **then return** *Reachable,* σ
9 $\quad\quad$ $PSS :=$ OrderAndFilter(PSS) ➔ Algorithm 1.11
10 $\quad\quad$ queue $:=$ queue $\cup$ PSS
11 $\quad$ **else if** e *is a partial solution* $ps = (\mathcal{C}, x, \sigma, r)$ **then**
12 $\quad\quad$ result, $ps^* :=$ FilterTInv(ps) ➔ Algorithm 1.5
13 $\quad\quad$ **if** result *is no skip* **then** result, $ps^* :=$ FilterTInvRemainder(ps) ➔ Algorithm 1.10
14 $\quad\quad$ **if** result *is skip* **then**
15 $\quad\quad\quad$ queue $:=$ queue $\cup$ BetterIntermediate(ps, m_0) ➔ Algorithm 1.6
16 $\quad\quad\quad$ queue $:=$ queue $\cup$ DistantInv(ps, ps^*, m_0) ➔ Algorithm 1.9
17 $\quad\quad$ **else**
18 $\quad\quad\quad$ $ps := (\mathcal{C} :=$ TransformJumps$(\mathcal{C}, x), x, \sigma, r)$ ➔ Algorithm 1.3
19 $\quad\quad\quad$ $\mathcal{C}' := \mathcal{C} \cup$ IncrementConstraints(ps, m_0) ➔ Algorithm 1.4
20 $\quad\quad\quad$ $\mathcal{C}' := \mathcal{C}' \cup$ InhibitorIncrementConstraints(ps, m_0) ➔ Algorithm 1.8
21 $\quad\quad\quad$ **if** $\mathcal{C}' \neq \mathcal{C}$ *and a solution* x *exists for* $\mathcal{C}'$ **then** queue $:=$ queue $\cup \{(x, \mathcal{C}')\}$
22 **if** *no partial solution was skipped* **then return** *Not reachable*
23 **else return** *Inconclusive*

If the current element is a partial solution, we first check if it can be skipped (Section 1.1.4.8). Later, we present a further criterion for skipping (Section 1.2.3.3). If the partial solution was skipped, we search for better intermediate markings and add them to the queue. Later, we present an approach to find new solutions by adding so-called “distant” invariants (Section 1.2.3). If the partial solution is not skipped, we first transform jumps and then generate increment constraints. If the net contains inhibitor arcs, an algorithm to be presented in Section 1.2.2 can add further constraints. If new constraints are found ($\mathcal{C}' \neq \mathcal{C}$) and there is a solution, it is added to the queue.

Finally, if the queue is empty and no full solution was found, the algorithm terminates with a conclusive “not reachable” answer if and only if no partial solutions were skipped during the process.

1.1.4.10 A Complex Example

Example. *As a complex example illustrating the whole algorithm, consider the Petri net PN in Figure 1.8a with the reachability problem $(1,0,0,2) \in R(PN, (0,0,0,2))$, i.e. to produce a token in p_0. The solution space (including partial solutions) is presented in Figure 1.9, and it is explained thoroughly in the following.*



(a) Petri net with the reachability problem to produce a token in p_0.

(b) Dependency graph of ps_0.

(c) Dependency graph of ps_1.

Figure 1.8: A complex example demonstrating various aspects of the algorithm.

The root of the solution space is the minimal solution vector $(1, 0, 0, 0, 0)$*, denoted by* sv_0 *(i.e. firing* t_0*). Since* t_0 *is not enabled, the only partial solution is* ps_0*, with the empty firing sequence* σ_0*. The algorithm builds a dependency graph (Figure 1.8b) to determine the increment constraints. The graph has edges from* p_1 *and* p_2 *to* t_0 *because they disable* t_0*. Edges in the opposite direction are not present, since firing* t_0 *does not increase the token count of* p_1 *or* p_2*.*

There are two source SCCs for ps_0*:*

- $SCC_1(\{p_1\}, \emptyset, \{t_0\})$*: One token is required in* p_1*, where* t_2 *can produce tokens, so the increment constraint is* $|t_2| \geq 1$*.*
- $SCC_2(\{p_2\}, \emptyset, \{t_0\})$*: One token is required in* p_2*, where* t_1 *and* t_4 *can produce tokens, so the increment constraint is* $|t_1| + |t_4| \geq 1$*.*

The new minimal solution fulfilling the state equation and the constraints is $(1, 1, 1, 0, 0)$*, labeled by* sv_1 *(i.e. the T-invariant* $\{t_1, t_2\}$ *is added). Since none of the transitions* t_0, t_1, t_2 *is enabled, the only partial solution is* ps_1 *with the empty firing sequence* σ_1*. The dependency graph for* ps_1 *can be seen in Figure 1.8c. There are two edges going from transitions to places as well, since* t_1 *and* t_2 *can increase the token count of* p_2 *and* p_1*. The only source SCC is* $SCC(\{p_1, p_2\}, \{t_1, t_2\}, \{t_0\})$*. One token in* p_1 *or* p_2 *might enable all the transitions of the SCC. The increment constraint takes the form* $|t_4| \geq 1$*, since* t_4 *is the only transition outside the remainder that can produce tokens in the SCC.*

The new solution vector is $(1, 1, 1, 1, 1)$*, denoted by* sv_2 *(i.e. the T-invariant* $\{t_3, t_4\}$ *is added). Two partial solutions can be found for* sv_2*:*

- ps_{21} *has the firing sequence* $\sigma_{21} = t_4t_3$*, but it is skipped by the T-invariant filtering optimization: it has the same remainder as* ps_1 *and the firing sequences are only different in the T-invariant* $\{t_3, t_4\}$*.*[5] *However, if only* t_4 *is fired from* $\sigma_{21} = t_4t_3$*, we are closer to enabling* t_0*, since it misses only one token. This better intermediate state is denoted by* bs_1*. In* bs_1*, one token is missing from* p_1*, where only* t_2 *could produce tokens, but* $r(t_2) > 0$*, so this partial solution cannot be extended with constraints.*
- *The other partial solution is* ps_{22} *with the firing sequence* $\sigma_{22} = t_4t_2t_1t_3$*.* ps_{22} *is also skipped by the T-invariant filtering optimization: it has the same remainder as* ps_0 *and the firing sequences are only different in the T-invariant* $\{t_1, t_2, t_3, t_4\}$*. However, there is a better intermediate state* bs_2*, where only* t_4 *and* t_2 *is fired from* σ_{22}*. This intermediate state misses a token from* p_2*, where* t_1 *and* t_4 *can produce tokens. Since* $r(t_1) > 0$*, the constraint is* $|t_4| \geq 2$*.*

[5] Without the optimization, the algorithm would add the T-invariant $\{t_3, t_4\}$ to the solution vector again and again. In this particular case, this would lead to a full solution, but in general, adding the same invariant infinitely many times can lead to non-termination.

The new solution vector is $(1, 1, 1, 2, 2)$*, denoted by* sv_3 *(i.e. the T-invariant* $\{t_3, t_4\}$ *is added).* sv_3 *has many partial solutions, but there is a full solution as well:* ps_3 *with the firing sequence* $\sigma_3 = t_4t_4t_2t_0t_1t_3t_3$*.*

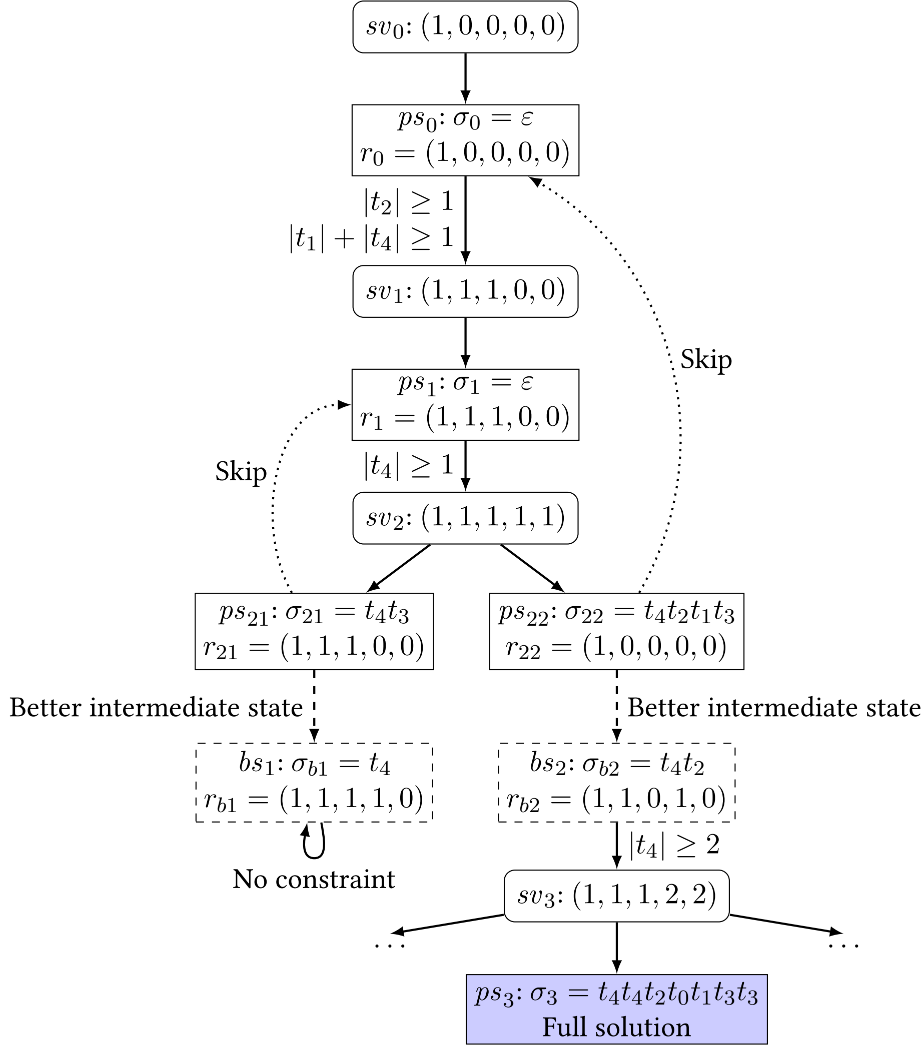


Figure 1.9: Solution space of the example seen in Figure 1.8. Solution vectors are denoted by rounded rectangles with their corresponding partial solutions in regular rectangles.

1.1.4.11 Completeness, Soundness and Conclusive Answers of the Algorithm

After Wimmel and Wolf published their algorithm, we examined if it is complete and sound, and whether it gives a conclusive answer[6] for every input [j1]. We conclude this section by summarizing these findings as they are relevant for the new contributions.

[6] In our papers [c4; j1; c5] we use the term "complete" to refer to an algorithm that always returns a conclusive answer. Furthermore, we use the term "correctness" for being both sound and complete.

Completeness. The algorithm is complete (does not report "reachable" for an unreachable state) by construction because a solution x is only claimed to be realizable if there is a corresponding full solution. The firing sequence σ of the full solution ($\wp(\sigma) = x$) is computed by exploring the concrete state space of the Petri net (Algorithm 1.1). Therefore, σ is realizable in the Petri net and it leads from the initial marking m_0 to the target marking m', proving the correctness of the answer "reachable".

Soundness. We proved by a counterexample[7] that the algorithm is unsound due to an overestimation in the increment constraint generating heuristic, resulting in an answer "not reachable" for a reachable marking [c4]. We suggested a method to detect such situations and to give an inconclusive answer (instead of a wrong "not reachable" answer). We also presented an extension to the algorithm that tries to find the solution in such cases, increasing the number of potential conclusive answers [j1]. To the best of our knowledge, our extended algorithm gives an inconclusive answer instead of "not reachable" if no full solution was found and potentially unsound optimizations or heuristics were applied to cut the search space. However, we did not prove this formally, leaving it as potential future work. Coming up with such a proof can be challenging due to the complexity of the algorithm, but a first step towards this direction could be done by restricting the proof to certain subclasses of nets that are easier to manage (e.g. acyclic nets [Mur89]).

Conclusive answers. We presented several subclasses of Petri nets for which the algorithm could not decide reachability, and we suggested solutions to most of them. However, we proved that the improved algorithm can still give inconclusive answers due to its iteration strategy.[8] In our current work we present a similar, but simpler proof (Section 1.2.3.1) and we propose a new iteration strategy to extend the set of problems where the algorithm is conclusive (Section 1.2.3.2).

1.2 Extensions

In this section, we propose various extensions and improvements to the CEGAR approach [WW11] presented before. We generalize the algorithm to be able to solve reachability problems with predicates (Section 1.2.1) and to handle inhibitor arcs (Section 1.2.2). We define the concept of distant invariants and a new iteration strategy, which extends the set of decidable problems (Section 1.2.3). Finally, we discuss different search strategies and propose an efficient combination (Section 1.2.4).

1.2.1 Reachability of Predicates

In Section 1.1.2, we generalized the reachability problem to predicates of the form $Am' \geq b$, where A is a matrix, and b is a vector of coefficients. However, as argued in Section 1.1.3 additional operators (such as "$=$" or "$\leq$") can be reduced to this form. We use "$\geq$" to make our presentation simpler, but in our implementation we provide all the operators that the ILP solver supports (and even their mixture, i.e. possibly different operators for each row of A).

[7] This counterexample on soundness was found by a fellow student, Zoltán Mártonka [c4].
[8] This example with an inconclusive answer was found by András Vörös [Vör18].

Predicates are a generalization of the reachability problem,[9] having various advantages. For example, it is easier to check the reachability of local conditions in complex nets because the target marking does not need to be fully specified. Furthermore, we can formulate properties involving multiple places, such as the sum of tokens.

Our key idea of handling predicates in the CEGAR approach is to transform the conditions over places into conditions on transitions. However, first, we have to make sure that the constraints do not allow negative tokens in any of the places, i.e. $m'(p_i) \geq 0$ must always implicitly hold for each $p_i \in P$. This can be done by extending the rows of A with a unit (identity) matrix of size $|P|$ and extending b with $|P|$ additional zeros. Let this extended matrix and vector be denoted by A' and b', respectively.

The second step is to substitute m' in the predicate with the state equation $m_0 + Cx = m'$. This yields the inequality of the form

$$A'(m_0 + Cx) \geq b',$$

which can be reordered in the form

$$(A'C)x \geq b' - A'm_0.$$

This set of inequalities can now be solved as an ILP problem for transitions. Note that the above substitution and reordering is also valid for operators other than "$\geq$". Our extended algorithm uses this modified form of the state equation as the initial abstraction ($\mathcal{C}_0$ in Algorithm 1.7), and expands it with additional (jump or increment) constraints during refinement.

Note that the solution space looks different with predicates than with basic reachability because there might be multiple potential target markings satisfying the constraints. In this generalized solution space, increment constraints do not necessarily add T-invariants. However, the algorithm does not rely on this fact and works for this generalized case as well.

Example. *Recall the Petri net PN in Figure 1.3 and suppose that we want to get at least one token in p_0. This can be formulated with $A = [1\ 0\ 0\ 0]$ and $b = (1)$. A' is obtained by extending A with a 4×4 unit matrix and b' becomes $(1, 0, 0, 0, 0)$. The base solution is $(1, 0, 0, 0, 0)$, i.e. firing t_0 which is not realizable because p_2 lacks a token. The algorithm adds a constraint $|t_1| \geq 1$ resulting in the new solution $(1, 1, 0, 1, 0)$, i.e. t_1 and t_3 are added (which is not a T-invariant). Note that the increment constraint only specified that t_1 has to be included, but t_3 also gets involved due to non-negativity constraints. This solution is realizable by $t_3 t_1 t_0$.*

1.2.2 Inhibitor Arcs

We presented inhibitor arcs in Section 1.1.1 as an important extension to Petri nets. They lift the expressive power to be Turing-complete by allowing to test for the lack of tokens at a place [Pet81]. While analysis methods are usually theoretically limited if inhibitor arcs are involved [Bus02] (e.g. reachability is undecidable in general [Chr99]), they are still relevant for practical cases.

The key challenge of handling inhibitor arcs in the CEGAR approach is that they do not appear in any form in the state equation, which is used as an abstraction. Therefore, a solution vector may be unrealizable because inhibitor arcs disable some transitions. In this case, tokens must be removed

[9] A reachability problem $m' \in R(PN, m_0)$ can be expressed with predicates by setting A to be a unit (identity) matrix, $b = m'$ and using the "=" operator in the constraints. In our prior work [c4; j1] we referred to this problem as "submarking coverability", but that terminology is misleading as coverability is not a generalization of reachability. Predicates are trivially a generalization of coverability (by setting A as the unit matrix and b as the marking to be covered), but as argued before, they are also a generalization of reachability.

from connected places. Our strategy is to add transitions to the solution vector, which consume tokens from such places. Increment constraints (Algorithm 1.4) are suitable for this purpose, but they have to be generated in a different way (Algorithm 1.8).

Algorithm 1.8: Find increment constraints for a partial solution with inhibitor arcs.

input : $ps = (\mathcal{C}, x, \sigma, r)$: Partial solution with $r > 0$
m_0: Initial marking
output: $\mathcal{C}'$: Constraints extended with new increment constraints

1 $\hat{m} \coloneqq$ final marking of ps, i.e. $m_0[\sigma\rangle\hat{m}$
2 // Step 1: build dependency graph G
3 $T_0 \coloneqq \{t \in T \mid r(t) > 0 \wedge \exists p \in P : (p, t) \in I \wedge \hat{m}(p) > 0\}$
4 $P_0 \coloneqq \{p \in P \mid \exists t \in T_0 : (p, t) \in I \wedge \hat{m}(p) > 0\}$
5 $E \coloneqq \{(p, t) \in P_0 \times T_0 \mid (p, t) \in I \wedge \hat{m}(p) > 0\} \cup \{(t, p) \in T_0 \times P_0 \mid w^+(p, t) < w^-(p, t)\}$
6 $\mathcal{C}' \coloneqq \mathcal{C}$
7 **foreach** *source* SCC_i *in* $G = (P_0 \cup T_0, E)$ **do**
8 $\quad P_i \coloneqq SCC_i \cap P_0$
9 $\quad T_i \coloneqq SCC_i \cap T_0$
10 $\quad X_i \coloneqq \{t \in T_0 \setminus SCC_i \mid \exists p \in P_i : (p, t) \in E\}$
11 $\quad$ // Step 2: estimate n
12 $\quad$ **if** $T_i \neq \emptyset$ **then** $n \coloneqq \min_{t \in T_i}(\sum_{p \in P_i \mid (p,t) \in I} \hat{m}(p))$
13 $\quad$ **else** $n \coloneqq \hat{m}(p_i)$ where p_i is the single place of P_i // if $T_i = \emptyset$ then $|P_i| = 1$
14 $\quad$ // Step 3: create constraint c'
15 $\quad P' \coloneqq P_i$
16 $\quad T' \coloneqq \{t \in T \mid r(t) = 0 \wedge \sum_{p \in P'}(w^+(p, t) - w^-(p, t)) < 0\}$
17 $\quad c' \coloneqq \sum_{t \in T'} \sum_{p \in P'}(w^-(p, t) - w^+(p, t)) \cdot |t| \geq$
$\quad\quad n + \sum_{t \in T'} \sum_{p \in P'}(w^-(p, t) - w^+(p, t)) \cdot \wp(\sigma)(t)$
18 $\quad \mathcal{C}' \coloneqq \mathcal{C}' \cup \{c'\}$
19 **return** $\mathcal{C}'$

The first step is to construct a dependency graph similar to the original one (Section 1.1.4.7). The graph consists of transitions that could not fire due to inhibitor arcs and places that disable these transitions (by having nonzero tokens). The arcs of the graph have an opposite meaning: an arc from a place to a transition means that the place disables the transition, while the other direction means that firing the transition would decrease the number of tokens in the place. Each source SCC of the graph is interesting because tokens cannot be consumed from them by another SCC.

The second step is to estimate the minimal number of tokens to be removed from each source SCC. There are two sets of transitions as well, $T_i \subseteq T$ and $X_i \subseteq T$. If $T_i \neq \emptyset$, then enabling one transition in T_i may enable all the others. In this case n is determined by finding the transition that has the least tokens in places connected with inhibitor arcs. Otherwise, if $T_i = \emptyset$, then P_i can only consist of a single place p_i, from which all the tokens should be removed.[10]

The third step is to construct an increment constraint for each source SCC, by firing transitions (with $r(t) = 0$) to consume the required number of tokens from the place of the SCC.

[10]This step of the algorithm is simpler than the corresponding part of the algorithm for normal arcs (Algorithm 1.4), because inhibitor arcs do not have weights. It is also possible to define inhibitor arcs with weights, in which case Algorithm 1.8 would have to adapt the more general solution of Algorithm 1.4.

When a partial solution is not a full solution, and there are transitions disabled by inhibitor arcs in the final marking, the previous heuristic is used to generate the constraint. If there are transitions disabled by normal arcs as well, both the original heuristic (Algorithm 1.4)[11] and the modified version (Algorithm 1.8) must be used, taking the union of the generated constraints (see main loop in Section 1.1.4.9). Analogously to the original heuristic, if n tokens have to be removed, the estimate is between 1 to n and can be repeated if too low.

Inhibitor arcs also affect the T-invariant filtering optimization (Section 1.1.4.8): an intermediate marking is now of interest when it has fewer tokens in a place, which is connected by an inhibitor arc to a transition that cannot fire. Formally, the definition of better intermediate markings (Definition 1.6, and also Algorithm 1.6) changes as follows.

> **Definition 1.7 (Better intermediate marking, inhibitor arcs).** An intermediate marking m_i is considered better than the final marking $\hat{m}$ of the firing sequence σ if Definition 1.6 holds *or* there exists a transition t with $r(t) > 0$ and a place p with $(p,t) \in I$ for which $\hat{m}(p) > 0 \land m_i(p) < \hat{m}(p)$ holds.

Example. *Consider the Petri net PN in Figure 1.10 with the reachability problem $(1,0,0,1) \in R(PN,(0,0,0,1))$, i.e. producing a token in p_0. The minimal solution is $(1,0,0,0,0)$, i.e. firing t_0, but it is not realizable. The original heuristic can determine that t_0 is disabled by p_1 and generates the constraint $|t_1| \geq 1$. Our extended heuristic continues by creating a dependency graph for inhibitor arcs, consisting of t_0 and p_3 with an edge from p_3 to t_0. The only source SCC is the single node p_3 from where t_3 can remove tokens so the constraint $|t_3| \geq 1$ is generated. The two constraints together force the ILP solver to add invariants $\{t_1,t_2\}$ and $\{t_3,t_4\}$ to the new solution $(1,1,1,1,1)$ which is realizable by $t_3t_1t_0t_2t_4$.*

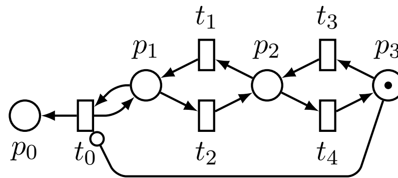


Figure 1.10: Example net where producing a token in p_0 requires increment constraints both to produce a token in p_1 and to remove one from p_3.

1.2.3 Distant Invariants

In this section, we show that the algorithm of Wimmel and Wolf [WW11] cannot decide reachability for relatively simple examples, because not every necessary invariant is explored (Section 1.2.3.1). We propose a new iteration strategy to traverse the invariant space by involving so-called “distant” invariants (Section 1.2.3.2). We show that this new approach extends the set of decidable problems, and we also give theoretical results on its limitations. We also present a new filtering criterion (Section 1.2.3.3), which can further avoid non-termination of the algorithm.

[11] Algorithm 1.4 also needs a slight modification. In step 1, T_0 needs the extra condition $\exists p \in P : \hat{m}(p) < w^-(p,t)$ to make sure that transitions in T_i are disabled by normal arcs.

1.2.3.1 Proof of Inconclusive Answers

We prove that the algorithm published by Wimmel and Wolf [WW11] can give an inconclusive answer with the following example. Consider the Petri net PN in Figure 1.11 with the reachability problem $(1,1,0) \in R(PN,(0,1,0))$, i.e. producing a token in p_0. The vector $x_s = (1,1,1,1,1)$ is a solution, realized by the firing sequence $\sigma_s = t_3 t_1 t_0 t_2 t_4$.

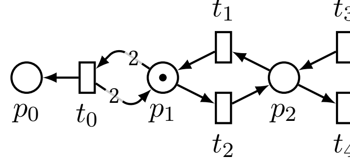


Figure 1.11: An example where checking if a token can be produced in p_0 results in an inconclusive answer by the algorithm.

The algorithm does the following steps. The minimal solution vector is $x_0 = (1,0,0,0,0)$, i.e. firing t_0. Since t_0 is not enabled, the only partial solution is $ps_0 = (\emptyset, x_0, \sigma_0 = \varepsilon, r_0 = (1,0,0,0,0))$. The algorithm finds that an additional token is required in p_1 and only t_1 can satisfy this need. With an increment constraint $c_1 : |t_1| \geq 1$, the T-invariant $\{t_1, t_2\}$ is added to the new solution vector $x_1 = (1,1,1,0,0)$. Only t_2 and t_1 can fire (in this order), thus the only partial solution for x_1 is $ps_1 = (\{c_1\}, x_1, \sigma_1 = t_2 t_1, r_1 = r_0)$. This partial solution is skipped by the T-invariant filtering optimization since the only difference from ps_0 is that all transitions of a T-invariant were fired. Furthermore, there are no better intermediate markings, since no additional token was "borrowed" from the T-invariant $\{t_1, t_2\}$. The algorithm terminates at this point, leaving the problem undecided. Without the filtering optimization, the algorithm would add the T-invariant $\{t_1, t_2\}$ again and again, preventing termination.

The problem is that the original algorithm does not recognize that although $\{t_1, t_2\}$ can fire, it only "circulates" the same token, instead of "lending" a new one. An extra token could be produced in p_2 (and then moved in p_1) using the T-invariant $\{t_3, t_4\}$. However, $\{t_3, t_4\}$ is not connected directly to p_1 (where the tokens are missing), so the iteration strategy of the algorithm does not try to involve it. We propose an extension to the iteration strategy in Section 1.2.3.2 in order to involve such "distant" invariants into the solution vector.

1.2.3.2 Involving Distant Invariants

Let y and z be T-invariants. We say that z is a distant invariant for y if z can produce tokens in a place connected to y. This can be written formally as follows.

> **Definition 1.8 (Distant invariant).** The T-invariant z is a distant invariant for the T-invariant y if a place p and transitions t_1, t_2 exist with $y(t_1) > 0$, $z(t_2) > 0$, $((t_1, p) \in E \vee (p, t_1) \in E)$, $w^+(p, t_2) - w^-(p, t_2) > 0$ and $y(t_2) = 0$.

The definition states that y includes t_1, z includes t_2 and t_1 is connected to a place p, where the firing of t_2 increases the number of tokens. This way, z can "borrow" tokens for y. The additional criterion $y(t_2) = 0$ is needed to ensure that we do not produce tokens for y by itself. In the example in Figure 1.11, $\{t_3, t_4\}$ is a distant invariant for $\{t_1, t_2\}$ because t_3 can produce tokens in p_2, which is connected to t_1 (and also to t_2).

When a transition in the remainder could not fire, the original algorithm tried to increase the token count on its input places. Our definition of distant invariants generalizes this concept in the following way. When a partial solution is skipped by the T-invariant filtering optimization, it means that a T-invariant was fired, but could not "lend" enough tokens to enable a transition in the remainder. The basic idea of involving distant invariants is to try to increase the token count in any place connected to the filtered invariant. If some tokens can be produced, the filtered invariant will then be able to transfer them indirectly to the place that lacks tokens. To achieve this, two main issues have to be addressed:

- Calculating the number of tokens to be produced for the invariant that was filtered.
- Avoiding non-termination if the distant invariant cannot help and would be added again.

Number of tokens. Estimating the required number of tokens is a hard problem, since the sum of the tokens in the places of a T-invariant may change during firing. Over-estimation can also be a problem: the final marking of the invariant may not be the "best" state regarding the number of tokens. Therefore, we produce only one token at a time and repeat this process if it was not enough.

Termination criterion. When a distant invariant does not help, there are two possible cases. The distant invariant z could (1) not lend any tokens to the filtered invariant y or (2) it could lend some, but not enough to enable a transition in the remainder.

The first case means that not only y lacks tokens, but z as well. Thus, we can now apply our strategy again, i.e. involving a distant invariant for $y + z$. This way, we form a "chain" of distant invariants, which is defined formally as follows.

> **Definition 1.9 (Chain of distant invariants).** Let $y_1, y_2, \ldots, y_n$ be T-invariants. We say that $y_1+y_2+\ldots+y_n$ is a chain of distant invariants if y_{i+1} is a distant invariant for y_i (for $1 \leq i < n$). A subchain of a chain $y_1 + y_2 + \ldots + y_n$ is a chain $y_1 + y_2 + \ldots + y_k$, with $k \leq n$.

The definition of distant invariants ensures termination for such chains since the newly involved distant invariant must have at least one transition that is not included in the previous ones, and the number of transitions in a Petri net is finite.

The second case indicates that z could lend some tokens, but not enough. Therefore, we can involve distant invariants again for y. If z is the only distant invariant for y, this simply results in adding z again, but in general, any distant invariant can be involved. However, if $y = y_1+y_2+\ldots+y_n$ is a chain, this would only produce tokens in places connected to y_n. Thus, we have to involve a distant invariant for every subchain in order to transfer the tokens to the originally filtered invariant (y_1).

Our new ideas above are formulated in Algorithm 1.9. The input of the algorithm is a partial solution ps' that was skipped due to ps and the initial marking m_0. Partial solutions are extended to store a chain of distant invariants (denoted by chainof), which is initially 0 (empty).

At first, we compute the difference between the solution vectors of ps and ps', we initialize the list of constraints with the constraints of ps', and calculate the number of better intermediate markings n_b.[12] The following two cases are possible.

- If the chain of ps is not empty, some distant invariants were already involved. If there are better intermediate markings ($n_b > 0$), then these invariants helped (but not enough) to enable a transition in the remainder. In this case, we can involve them again, so the chain of ps' is the same as in ps, and we involve a distant invariant for every subchain.

[12] Better intermediate markings were already computed when the partial solution ps' was skipped (see main loop in Section 1.1.4.9). In the implementation, we cache their number (n_b) along with ps' to avoid double calculation.

- Otherwise, we extend the chain of ps with z and involve distant invariants only for the whole chain. However, we have to first check if z is really an extension to the chain of ps, since ps' can be a solution obtained by the original increment constraints.

Algorithm 1.9: Get new solution by involving distant invariants.

input : ps': Partial solution skipped
ps: Partial solution that caused skipping ps'
m_0: Initial marking

output: x: New solution vector (if found) by involving distant invariants

```
1  z := difference invariant between ps and ps'
2  C* := constraints of ps'
3  n_b := |BetterIntermediate(ps', m0)|   ➔ Algorithm 1.6
4  if chainof(ps) ≠ 0 and n_b > 0 then
5      chainof(ps') := chainof(ps)
6      for each subchain of chainof(ps') do
7          C* := C* ∪ {constraint to involve a distant invariant for the subchain}
8  else if z is a distant invariant for chainof(ps) or chainof(ps) = 0 then
9      chainof(ps') := chainof(ps) + z
10     C* := C* ∪ {constraint to involve a distant invariant for chainof(ps')}
11 x := solve the state equation with C*
12 return x
```

Finding a constraint to involve a distant invariant for a chain (or subchain) y is quite straightforward. We get the places $P' \subseteq P$ connected to the transitions of y, and we create a constraint using the third step of the increment constraint generating heuristic (Algorithm 1.4) to produce a token ($n = 1$) in these places. If no constraint can be found, the algorithm returns no new solution. Otherwise, we solve the state equation extended with $\mathcal{C}^*$ and return the solution (if found). If there are multiple distant invariants for y, all of them will be found by using jump constraints in the algorithm.

Example. *This new strategy can solve the example in Figure 1.11 trivially. As a complex example, consider the Petri net PN in Figure 1.12 with the reachability problem $(1, 1, 0, 0, 2) \in R(PN, (0, 1, 0, 0, 2))$, i.e. producing a token in p_0. The minimal solution is to fire t_0, but it is not enabled. Thus, the T-invariant $\{t_1, t_2\}$ is added twice in order to get two additional tokens in p_1. This invariant can fire, but it does not help to get closer to enabling t_0, so the partial solution is skipped without any better intermediate marking. At this point, our new algorithm tries to produce a token in any of the places connected to $\{t_1, t_2\}$, i.e. p_1 and p_2 by distant invariants. Therefore, the T-invariant $\{t_3, t_4\}$ is added once to the new solution. This invariant can also fire but does not help to enable t_0. The partial solution is skipped, and since $\{t_3, t_4\}$ is a distant invariant for $\{t_1, t_2\}$, the algorithm now tries to produce a token in places connected to the chain $\{t_1, t_2\} \cup \{t_3, t_4\}$, i.e. in p_1, p_2, and p_3. This implies that the invariant $\{t_5, t_6\}$ is added once. Firing this invariant does not enable t_0, but yields an extra token in p_1, which is a better intermediate marking. Thus, the partial solution is skipped but the algorithm now tries to involve distant invariants for every subchain, namely for $\{t_1, t_2\}$ and $\{t_1, t_2, t_3, t_4\}$, resulting in the addition of $\{t_3, t_4\}$ and $\{t_5, t_6\}$. The solution vector is now $(1, 2, 2, 2, 2, 2, 2)$, which can be realized by the firing sequence $t_5 t_5 t_3 t_3 t_1 t_1 t_0 t_2 t_2 t_4 t_4 t_6 t_6$.*

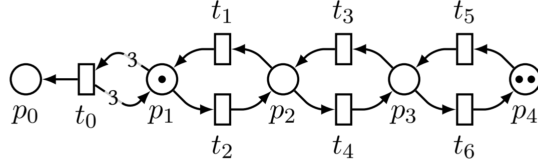
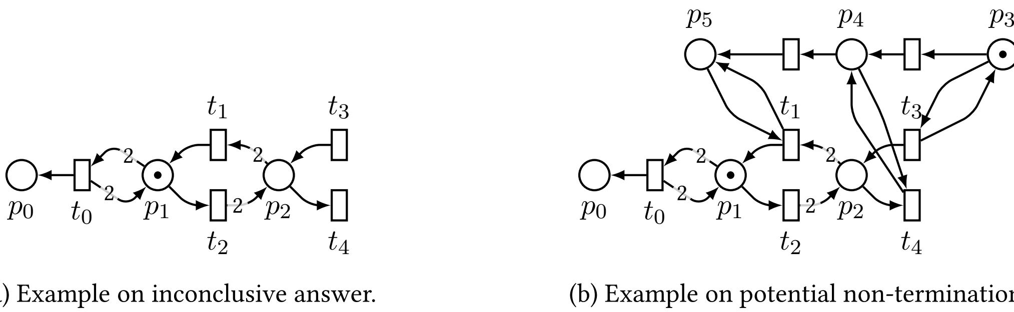


Figure 1.12: A complex example where distant invariants are discovered in multiple iterations.

Limitations. Although our new approach can solve a new range of problems, it also has some limitations in giving conclusive answers.



(a) Example on inconclusive answer.

(b) Example on potential non-termination.

Figure 1.13: Example nets illustrating the limitations of distant invariants.

Example. *Consider the Petri net PN in Figure 1.13a with the reachability problem $(1,1,0) \in R(PN,(0,1,0))$, i.e. producing a token in p_0. The minimal solution is firing t_0, which is not enabled. Thus, the T-invariant $\{t_1,t_2\}$ is added once in order to get an additional token in p_1. This invariant can fire, but it does not help to get closer to enabling t_0, so the partial solution is filtered. At this point the algorithm tries to produce tokens for $\{t_1,t_2\}$ using distant invariants, which implies adding $\{t_3,t_4\}$ once. This invariant can fire, lending a token in p_2. However, t_1 requires two tokens to fire and produce one in p_1. This partial solution is also filtered, and there are no better intermediate markings since we only count the tokens in places connected to the disabled transition t_0, which is p_1. The algorithm terminates at this point, leaving the problem inconclusive.*

A trivial idea for this example would be to extend the definition of better intermediate markings (Definition 1.6) to count tokens not only in places connected to the transition that cannot fire but in places connected to the filtered T-invariant as well. This can be formalized as follows. Let $ps = (\mathcal{C}, x+y, \sigma, r)$ be a partial solution that was skipped due to the invariant y. Suppose that we obtained $ps' = (\mathcal{C}', x+y+z, \sigma', r)$ by involving the distant invariant z for y, which could not enable any transition in the remainder, thus ps' is skipped as well. Furthermore, suppose that no better intermediate marking was found using Definition 1.6 (as in the example in Figure 1.13a). Given a partial solution ps and a place p let $\max(ps,p)$ be $\max(m(p))$ during firing σ of ps from the initial marking m_0. Then the definition of better intermediate markings can be generalized in the following way.

> **Definition 1.10.** Given the partial solutions ps and ps' as described above, an intermediate marking m_i of σ' is better than the final marking $\hat{m}$ if Definition 1.6 holds or a transition t with $y(t) > 0$ and a place p with $(p,t) \in E \vee (t,p) \in E$ exists for which $m_i(p) > \max(ps,p)$ holds.

The generalized definition states that the intermediate marking is also considered better if there is a place connected to the filtered T-invariant, which contains more tokens than in any marking in the firing sequence of the previous partial solution. If a better intermediate marking exists for ps' using this definition, then we can involve z again. However, this definition would often lead to non-termination since the filtered T-invariant (y) is already enabled (otherwise, it would not have been filtered). Thus, we cannot give an upper bound on the number of tokens in p, as opposed to our original definition, where we produce tokens in p until the transition that is disabled by p gets enabled.

Example. *Consider the Petri net PN in Figure 1.13b with the reachability problem $(1,1,0,0,0,1) \in R(PN,(0,1,0,1,0,0))$, i.e. producing a token in p_0 and moving the token from p_3 to p_5. This net works similarly to the net in Figure 1.13a, but occurrences of the transitions t_3, t_4, and t_1 can only appear in this order, due to the upper part (places p_3, p_4, p_5) of the net. This makes the target marking unreachable. As in the previous example, first $\{t_1, t_2\}$, then $\{t_3, t_4\}$ is added. Suppose now, that we consider it a better intermediate marking when t_3 produced a token in p_2. This implies that $\{t_3, t_4\}$ is added again. Now t_3 can fire two times, producing two tokens in p_2. There are two possible sequels. If t_1 fires, it produces an extra token in p_1 and enables t_0. However, the extra tokens must be consumed in order to reach the final marking, but t_4 cannot fire after t_1. The search terminates on this path since no more solutions can be found. The second case is that t_4 fires, which consumes the tokens from p_2 so t_1 cannot transfer them to p_1. Thus, t_0 is still not enabled, but we had a better intermediate state since we had two tokens in p_2. Therefore, $\{t_3, t_4\}$ is added again and this process repeats avoiding termination.*

The examples in Figure 1.13 show that the generalized definition (Definition 1.10) may help to decide reachability for some instances, but may also yield non-termination.

Remark. Distant invariants are automatically computed and added to the solution vector on the fly. However, in principle, it would be possible to compute and express them as a semi-linear space. Given a Petri net, all the (minimal) T-invariants can be calculated [MS82; CU05], and then for a given T-invariant x, Definition 1.8 can be transitively applied to find the (minimal) distant invariants. For example, if x and y are connected, and y and z are also, then y and $y+z$ (and their linear combinations $k_1 y + k_2(y+z)$) are distant invariants for x. Considering all distant invariants, they form a semi-linear set by taking the union of the linear spaces of distant invariants for each individual T-invariant.

1.2.3.3 New Filtering Criterion

When a partial solution is skipped using the T-invariant filtering optimization (Section 1.1.4.8), we may obtain new solutions from it through intermediate markings or distant invariants. This yields a new branch in the search space, which can also lead to non-termination.

There are special cases where T-invariants can either fire or not, both being a maximal firing sequence. As an example, consider the Petri net in Figure 1.12 and suppose that t_1, t_2, t_3, and t_4 each has to fire once. A possible maximal firing sequence is $t_2 t_4 t_3 t_1$, but $t_2 t_1$ is also maximal, since neither t_3 nor t_4 is enabled afterwards. When such invariants exist, it is possible that the following two partial solutions are obtained from $ps = (\mathcal{C}, x, \sigma, r)$ after adding the invariant y:

- $ps' = (\mathcal{C}', x+y, \sigma', r)$, with $\wp(\sigma') = \wp(\sigma) + y$, and
- $ps'' = (\mathcal{C}', x+y, \sigma, r+y)$.

In the first case, the invariant was fired (i.e. added to the firing sequence), while in the second case, it was not fired (i.e. added to the remainder). The first case can be detected by the T-invariant filtering optimization. However, we found that the second case can also lead to non-termination if there are at least two T-invariants with this property.

To overcome this problem, we detect when a T-invariant is added to the remainder, i.e. we get $ps'' = (\mathcal{C}', x+y, \sigma, r+y)$ from $ps = (\mathcal{C}, x, \sigma, r)$. However, ps'' cannot be filtered immediately because the remainder is different, so the abstraction refinement may add new invariants that can help. We only skip ps'' if ps was omitted by the original T-invariant filtering optimization, which also means that ps'' was obtained through intermediate markings or distant invariants. This approach is illustrated by Algorithm 1.10.

Algorithm 1.10: Check if a part. sol. can be filtered based on T-invariants in the remainder.

```
input  : ps'' = (C'', x'', σ'', r''): Partial solution to be checked
output: (skip or no skip, ps*): Result and the other part. sol. that caused skipping (if exists)
1 foreach solution vector x from ancestors of ps'' do
2   foreach partial solution ps* = (C, x, σ, r) of x do
3     (result, _) := FilterTInv(ps*)  ➔ Algorithm 1.5
4     if ps* ≠ ps'' and ℘(σ) = ℘(σ'') and r'' − r is T-inv. and result is skip then
5       return (skip, ps*)
6 return (no skip, ps'')
```

1.2.4 Hybrid Search

As already mentioned in Section 1.1.4, the algorithm of Wimmel and Wolf [WW11] traverses the semi-linear solution space of the state equation. At each non-realizable solution, multiple (jump and/or increment) constraints can be applied, each yielding a new path in the solution space. However, the authors did not publish the strategy for the solution space traversal [WW11]. An overview pseudo-code was published later [WW12] suggesting an ordering of solutions based on the sum of their elements. In this section we present three different search strategies: depth-first search (Section 1.2.4.1), breadth-first search (Section 1.2.4.2) and our new approach, a hybrid strategy (Section 1.2.4.3), which combines the advantages of DFS and BFS. Measurement results supporting our statements in this section can be found in Section 1.4.3.

1.2.4.1 Depth-First Search

Depth-first search (DFS) can be very effective regarding memory usage and computation time as well. It only stores one path of the solution space in memory at a time for backtracking purposes, and it has a fast convergence if several invariants have to be added to reach a realizable solution. However, DFS has some disadvantages as well:

- It may not find the minimal solution by choosing a path, which contains a solution but not the minimal one.
- It may fail to terminate in an infinite solution space by choosing a path, where T-invariants can be added infinitely many times without finding a realizable solution.

The T-invariant filtering optimization (Section 1.1.4.8) and our new filtering criterion (Section 1.2.3.3) can cut the search space, but do not always detect infinite loops. We tried to give stronger criteria for cutting, but then realizable solutions were lost, reducing the set of decidable problems.

1.2.4.2 Breadth-First Search

Due to the problems of DFS, we implemented a breadth-first search (BFS) version of the algorithm as well. The number of base solutions can grow exponentially, but it is always finite, so we still use DFS between the base solutions and only use BFS in the linear space of invariants. As opposed to DFS, it is less efficient but always finds the minimal solution if the target marking is reachable. When the target marking is not reachable, BFS may fail to terminate in an infinite solution space. The T-invariant filtering optimization can prevent this in some cases and can also make the computational time shorter.

1.2.4.3 Hybrid Search

We also developed a new, hybrid search strategy, which combines the advantages of DFS and BFS. We traverse the base solutions using DFS as previously. When exploring the invariant space over a base solution our main strategy is DFS, but with a little BFS-like extension: at each solution x, we generate all partial solutions belonging to x (instead of continuing the search with the first one) and filter them based on a partial order.

Ordering of partial solutions. We define an ordering over vectors and partial solutions as follows.

> **Definition 1.11 (Ordering of vectors).** A vector x is less than a vector y (denoted by $x < y$), if and only if $x(i) \leq y(i)$ for each index i and $x \neq y$.

> **Definition 1.12 (Ordering of partial solutions).** A partial solution $ps_1 = (\mathcal{C}, x, \sigma_1, r_1)$ is less than a partial solution $ps_2 = (\mathcal{C}, x, \sigma_2, r_2)$ (denoted by $ps_1 < ps_2$), if and only if $r_2 < r_1$.

A partial solution ps_1 is less than a partial solution ps_2 if the remainder r_2 is less than r_1. This means that ps_2 is closer to realization, since every transition fired in the sequence of ps_1 was also fired in ps_2, but ps_2 may have more fired transitions. Note that this is a partial order, since partial solutions ps_1, ps_2 may exist with $ps_1 \not< ps_2$ and $ps_2 \not< ps_1$, e.g. if $r_1 = (1, 0)$ and $r_2 = (0, 1)$. Furthermore, the partial order is only defined over partial solutions belonging to the same solution vector x (and constraints $\mathcal{C}$). This is sufficient, because – as described in the following – we will only compare such partial solutions.

Filtering partial solutions. For our filtering criterion, we define maximal and minimal partial solutions with respect to a given solution vector x as follows.

> **Definition 1.13 (Maximal partial solution).** A partial solution ps of a solution x is maximal, if and only if no other partial solution ps' exists for x with $ps < ps'$.

> **Definition 1.14 (Minimal partial solution).** A partial solution ps of a solution x is minimal, if and only if no other partial solution ps' exists for x with $ps' < ps$.

The filtering criterion is quite simple; we only keep minimal and maximal partial solutions, as illustrated by Algorithm 1.11 (see also the main loop in Section 1.1.4.9). Not that since the ordering is partial, there can be multiple minimal and maximal partial solutions.

Algorithm 1.11: Filter a set of partial solutions based on partial order.

input : PSS: Set of partial solutions (belonging to same solution vector)
output: $PSS' \subseteq PSS$: A subset of partial solutions filtered by the partial order
1 **return** $\{ps \in PSS \mid \nexists ps' \in PSS : ps < ps' \vee ps' < ps\}$

We keep the maximal partial solution because it has a minimal remainder, i.e. it is the closest to realizing the solution vector. Also, the T-invariant filtering optimization works well for maximal partial solutions, since every T-invariant that can fire, must also fire (i.e. it is added to the firing sequence). A minimal partial solution has a maximal remainder, i.e. not every enabled T-invariant was fired. This yields a slower convergence to a realizable solution. However, since the remainder is different from the remainder of the maximal partial solution, the abstraction refinement may involve different invariants.

1.3 Implementation

We implemented the original algorithm [WW11] and our new contributions (Section 1.2) as a plug-in for the PetriDotNet framework [c7]. Version 1.5 of PetriDotNet includes our new contributions and is publicly available online.[13] An overview of the architecture can be seen in Figure 1.14.

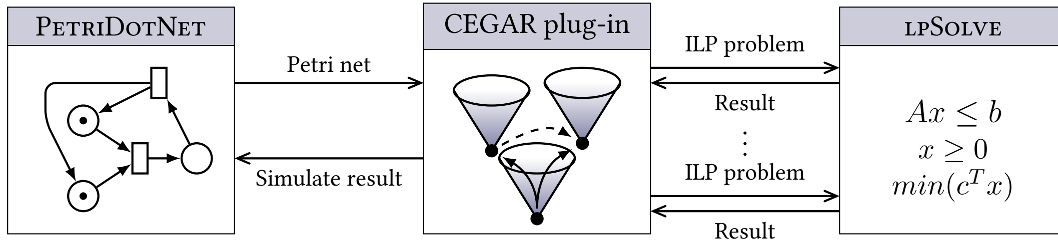


Figure 1.14: Overview of the architecture. The CEGAR plug-in is implemented as an add-on to the PetriDotNet framework. The plug-in relies on lpSolve to solve ILP queries.

PetriDotNet is an application for editing, simulating, and analyzing Petri nets. It has been extensively used in both education and industrial use cases [j2]. PetriDotNet is written in C# and provides a base library, and a user interface for editing and simulating Petri nets. Analysis algorithms (such as our CEGAR approach) are implemented via the plug-in interface of PetriDotNet.

When the CEGAR plug-in is started, it gets a reference to the current Petri net. The plug-in provides a graphical user interface to configure its parameters (e.g. target marking, optimizations). After setting the parameters, the plug-in starts to explore the solution space using an ILP solver. We used lpSolve[14] for this purpose as it is freely available and provides an easily usable interface for C#. The abstraction is refined in an iterative process, so the CEGAR plug-in may call the solver several times. If a realizable solution is found, the firing sequence can be simulated visually in PetriDotNet.

[13] http://petridotnet.inf.mit.bme.hu/en/
[14] http://sourceforge.net/projects/lpsolve/, version 5.5

1.4 Evaluation

In this section we evaluate our extensions and contributions presented before (Section 1.2). We formulate the following three research questions for the evaluation.

RQ1 How does the algorithm (including our new contributions) scale for Petri nets with an increasing parameter (affecting the size of the structure and/or the state space)?
RQ2 How does our implementation perform compared to other tools and algorithms?
RQ3 How do the different search strategies perform compared to each other?

Table 1.1: Summary of the evaluation goals, the models used and the observed output.

	Goal	Models	Observed
RQ1	Scalability with parameter	MCC	Runtime
RQ2	Comparison to other tools	MCC, custom	Runtime, decidability
RQ3	Comparison of search strategies	Custom	Runtime, cost

Table 1.1 summarizes the evaluation goals, the models used, and the observed output. In RQ1 (Section 1.4.1), we observe the runtime of the algorithm with respect to an increasing parameter value on models from the Model Checking Contest (MCC) [Kor+12]. The parameter can affect the size of the net structure or the size of the state space. RQ2 (Section 1.4.2) compares our implementation to other tools and algorithms on MCC models, including the original implementation [WW11] and a variant of saturation [VDB11]. Besides runtime, the decidability of the problem is also observed here, for which we also use further custom models (related to distant invariants). Finally, RQ3 (Section 1.4.3) uses custom models with large solution space to compare the different search strategies presented in Section 1.2.4. Besides runtime, we also measure the cost of the solution (length of the firing sequence).

1.4.1 RQ1: Scalability

This section presents how the runtime of the algorithm scales for several models with a given parameter. For more details on the models, the interested reader is referred to the provided references and Appendix B of [Dar14]. Measurements were executed on a laptop with Intel Core i5 M430 2.27 GHz processor, 3 GB RAM, and Windows 7 x32. The algorithms used DFS for this research question.

Counter. The counter model [CZJ12] represents a simple n bit binary counter with $|P| = n$ places for each bit, $|T| = n + 1$ transitions, and $|E \cup I| = (n^2 + 5n)/2$ arcs for changing each bit and resetting the whole counter. This model contains $|I| = n$ inhibitor arcs. The problem solved by the algorithm is to count from 0 to $2^n - 1$, where n is the parameter. The results can be seen in Figure 1.15. The runtime clearly scales exponentially with the parameter, which is not surprising, since the length of the firing sequence solving the problem is also exponential of n.

Dining philosophers. The dining philosophers model [Dij71] is often used to illustrate the problems of parallel programming and mutual exclusion. There are n philosophers around a circular table. Each philosopher has a plate, and there is a fork between every two plates. A philosopher can eat if he has a fork in both hands. Since two neighbors share a fork, at most $\lfloor n/2 \rfloor$ philosophers can eat at the same time. Each philosopher is either thinking or eating. If a philosopher gets hungry, he grabs the forks next to him and eats. After eating, he puts back the forks. There is a possibility for deadlock

if all the philosophers get hungry at the same time, and they all grab one fork. In this case, none of them can eat. Therefore, they will not put back the forks.

The problem solved by the algorithm is to reach a state where every second philosopher is eating. The results can be seen in Figure 1.16. This model has a large structure ($|P| = 6n, |T| = 4n, |E| = 14n$). Therefore, finding the solution vector with the ILP solver is already a hard problem.

Parameter	Runtime (s)
4	0.011
8	0.02
12	0.038
14	0.096
16	0.195
18	0.994
20	3.726

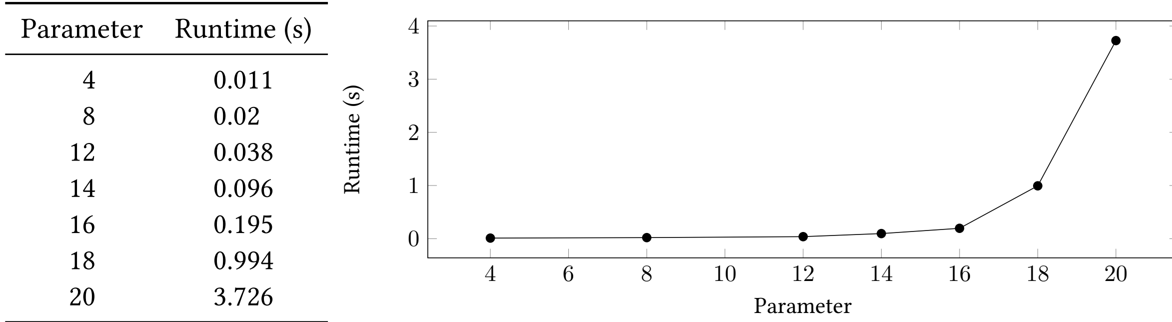


Figure 1.15: Measurement results for the counter model.

Parameter	Runtime (s)
10	0.092
20	0.241
30	0.488
50	1.468
100	9.754
200	83.357

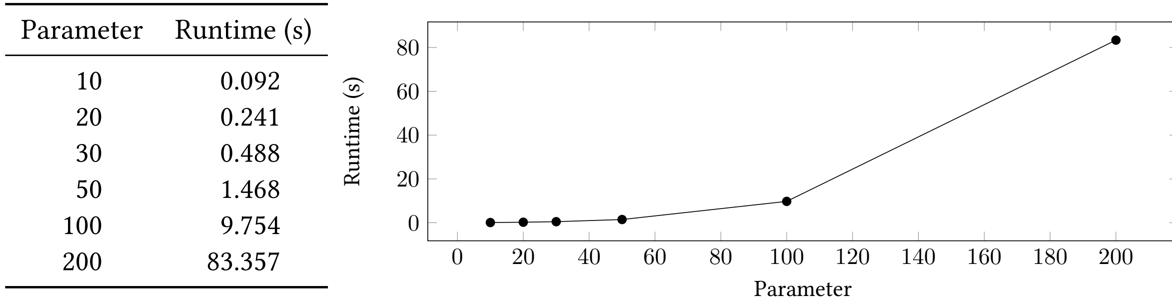


Figure 1.16: Measurement results for the dining philosophers model.

FMS. The FMS model [CT93] represents a flexible manufacturing system where different types of parts are assembled together. The parameter of this model is the number of parts to be assembled (n), which determines the initial marking (n tokens in three places). The structure of the net is the same for all n ($|P| = 22, |T| = 20, |E| = 50$). The results can be seen in Figure 1.17. Since the structure of the net does not change, the size of the abstract model is constant. However, the length of the firing sequence solving the problem grows linearly, which yields linear scalability for the runtime.

Kanban. The Kanban model [TTC96] represents a production scheduling method. The parameter of this model determines the initial marking (n tokens in four places), and the structure is the same ($|P| = |T| = 16, |E| = 40$) for all n. The results can be seen in Figure 1.18. Although the size of the model does not change, the runtime scales exponentially with the parameter. We experienced that the algorithm can find a realizable solution vector quickly. Still, it examines many partial solutions before it finds a full solution, i.e. there are many dead-ends in the partial solution tree.

Slotted ring. The slotted ring model [Pas+94] represents a network protocol. The parameter is the number of participants in the network. The results can be seen in Figure 1.19. Although the size of

the model grows ($|P| = |T| = 8n, |E| = 24n$), the ILP solver can handle this model well, and this yields a polynomial runtime.

Parameter	Runtime (s)
10	0.046
50	0.053
100	0.058
200	0.078
400	0.115
800	0.191
1600	0.321
3200	0.638
6400	1.28
12800	2.571

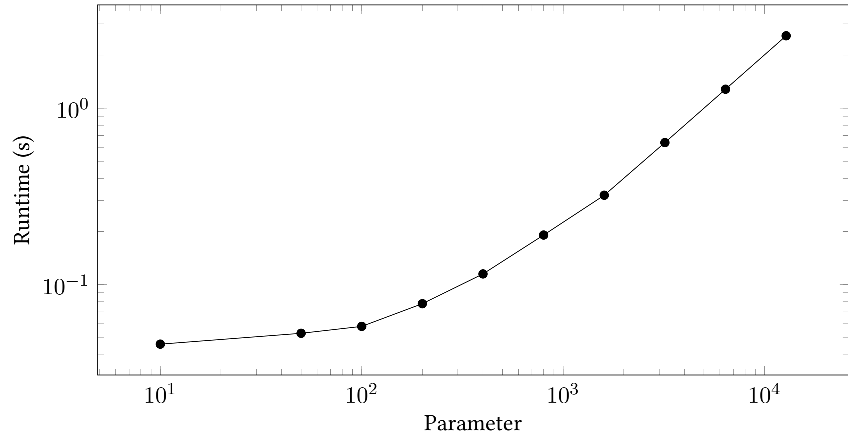


Figure 1.17: Measurement results for the FMS model.

Parameter	Runtime (s)
10	0.352
13	1.147
16	3.424
19	8.04
22	17.51
25	35.061
28	64.947

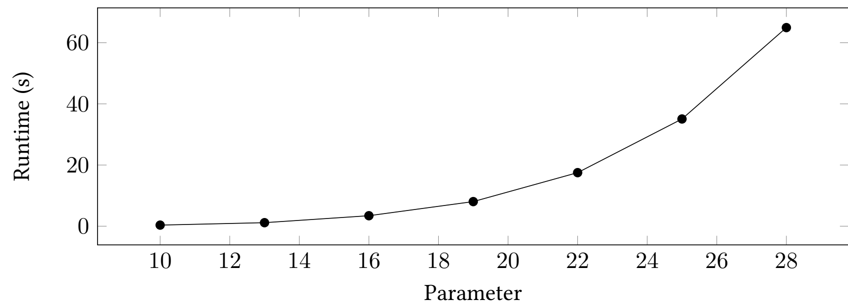


Figure 1.18: Measurement results for the Kanban model.

Parameter	Runtime (s)
10	0.207
20	0.571
30	1.206
35	1.585
50	3.461

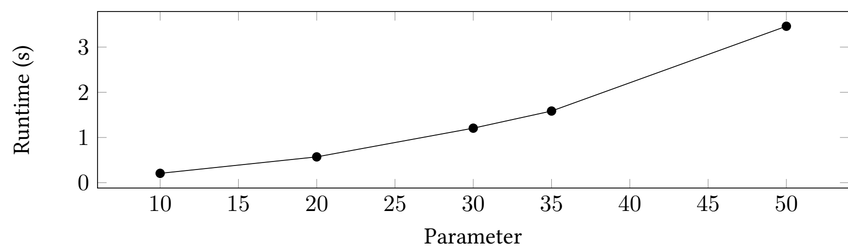


Figure 1.19: Measurement results for the slotted ring model.

Discussion. One interesting topic to discuss is the effect of the new contributions on the scalability of the algorithm. Reachability of predicates only modifies the initial set of constraints but has no effect on basic reachability queries. The heuristics for inhibitor arcs are only applied when the net does contain inhibitor arcs. When the net does not have inhibitor arcs, scalability is practically the same (apart from the negligible time spent on checking if the net contains inhibitor arcs). The distant

invariants are applied when partial solutions are skipped, possibly continuing the search on an otherwise terminated branch. While this can have a significant effect on scalability, it also helps to extend the set of problems where a conclusive answer is given. In the implementation, distant invariants can be turned on/off, so a potential trade-off can be to first run the algorithm without distant invariants, and only apply them if the answer is inconclusive. Finally, we observed that for these examples the hybrid search does not yield significant advantage so we performed standard DFS in this research question, and created custom models with large solution spaces for RQ3 (Section 1.4.3).

1.4.2 RQ2: Comparison to Other Tools and Algorithms

We compared our algorithm in PetriDotNet to the implementation of Wimmel and Wolf [WW11], which is called the Sara tool.[15] We also compared our approach to a variant of the well-known saturation-based model checking algorithm [CLS01] implemented in PetriDotNet [VDB11]. The results can be seen in Table 1.2, where *TO* refers to a timeout of 600 seconds, *ERR* means a runtime exception and *NS* implies that the algorithm terminated, but could not solve the problem. Note that in order to check reachability, saturation actually builds up the whole state space (symbolically) and could then answer additional queries.

The FMS model [CT93] represents a flexible manufacturing system. The parameter of the model determines the size of the state space, while the structure of the net is fixed. The results show that our algorithm outperforms both saturation and the Sara tool. The Kanban model [TTC96] illustrates a production scheduling method. The parameter determines the size of the state space. We experienced that our algorithm can find a realizable solution quickly, but it examines many partial solutions before finding the full solution. The Dining philosophers model [Dij71] is often used to show the problems of parallel programming and mutual exclusion. As the parameter grows, both the structure of the net and the state space becomes larger. Saturation and Sara perform better for these models.

The Distant$n$ models are built by us[16] to test our new iteration strategy, which involves distant invariants. The Distant1 and Distant3 models can also be seen in Figure 1.11 and 1.12 respectively. After publishing our former proof of inconclusive answers [j1], we contacted Wimmel and Wolf, and they also extended their implementation to be able to solve Distant1. However, the original algorithm cannot solve complex examples of distant invariants. As the state spaces of these models are infinite, saturation cannot handle these problems.

Due to the complexity of the models, further examination is required to determine how the structure and behavior of the models affect the performance of the algorithms and which algorithm is the most effective for a given type of models. This is an interesting future research direction.

1.4.3 RQ3: Comparison of Search Strategies

The solution space (i.e. the abstract model) is usually small for the examples presented in Table 1.2, so every search strategy has a similar performance. We created models[17] with many T-invariants (i.e. a large solution space) to evaluate the different search strategies. The results can be seen in Table 1.3, where the cost corresponds to the size of the solution, i.e. $\sum_{t \in T} x(t)$. The two parameters in the model name determine the number of invariants. The asterisk indicates a different ordering of places and transitions.

[15] http://www.service-technology.org/sara/index.html

[16] The Distant$n$ models are available at http://inf.mit.bme.hu/en/pn2015.

[17] The Chain n+k models are available at http://inf.mit.bme.hu/en/pn2015.

Table 1.2: Comparison of the execution time (s) of our implementation to Sara and saturation. *TO* indicates a timeout after 600 seconds, *ERR* means a runtime exception, *NS* denotes an inconclusive answer, while “-” represents an unsupported case.

Model	Our algo.	Sara	Saturation
FMS-10	0.041	0.001	0.06
FMS-50	0.048	0.018	1.09
FMS-100	0.056	0.059	8.03
FMS-200	0.071	0.278	69.7
FMS-400	0.105	0.868	TO
FMS-800	0.226	3.537	TO
FMS-1600	0.317	ERR	TO
FMS-3200	0.65	ERR	TO
FMS-6400	1.274	ERR	TO
FMS-12800	2.54	ERR	TO
Kanban-10	0.032	0.03	0.002
Kanban-13	1.074	0.05	0.003
Kanban-16	3.055	0.09	0.01
Kanban-19	7.128	0.134	0.03
Kanban-22	16.039	0.2	0.03
Kanban-25	31.181	0.268	0.05
Dphil-10	0.078	0.005	0.01
Dphil-20	0.204	0.012	0.02
Dphil-30	0.399	0.021	0.03
Dphil-50	1.156	0.037	0.03
Dphil-100	6.989	0.094	0.04
Dphil-200	67.603	0.33	0.05
Distant1	0.027	0.001	-
Distant2	0.068	NS	-
Distant3	0.083	NS	-
Distant4	0.116	NS	-
Distant5	0.078	NS	-
Distant6	0.063	NS	-
Distant7	0.137	NS	-

It is clear that DFS is more efficient than BFS regarding computational time. However, it often fails to find the minimal solution. Our hybrid strategy often outperforms DFS while also being closer to the minimal solution. The ordering of places and transitions can sometimes have a big impact on the execution time. This is not surprising for DFS as it has to pick and explore subtrees in some order. But this can also affect BFS and the hybrid search as in the implementation of Algorithm 1.1, if they find a full solution, they stop immediately and do not explore the rest of the partial solutions.

Table 1.3: Measurement results for different search strategies. The cost is the size of the solution found. *TO* refers to a timeout.

	DFS		BFS		Hybrid	
Model	Time (s)	Cost	Time (s)	Cost	Time (s)	Cost
Chain 1+2	0.04	7	0.055	7	0.039	7
Chain 1+3	0.095	13	0.828	13	0.1	13
Chain 1+4	0.291	21	85.24	21	0.288	21
Chain 1+4*	24.2	35	55.28	21	1.498	29
Chain 1+5	54.59	39	TO	31	56.36	39
Chain 2+2	0.076	11	0.277	11	0.074	11
Chain 2+3	0.197	19	12.768	19	0.288	23
Chain 2+3*	2.28	29	5.288	19	1.387	23

1.5 Related Work

Our contributions are direct extensions of the approach proposed by Wimmel and Wolf [WW11; WW12], improving its expressive power, and widening the set of decidable problems.

Reachability analysis of Petri nets is often performed by standard model checking techniques, including various decision diagrams [Pas+94; CT05; Kan+15; Amp+16], saturation [CLS01; VDB11; Mol19], and partial order reduction [Val91; Wol18]. Such techniques can also be applied in the CEGAR approach during the bounded exploration of the state space when looking for partial solutions for a given solution vector. We apply a variant of the stubborn set partial order reduction method in our approach [Sch99].

A key feature of Petri nets is the strong theory on their structural analysis [Wol19], which is less prone to state space explosion. The CEGAR approach also applies structural techniques, namely the state equation and T-invariants. Esparza and Melzer [EM00] verify safety properties (described by predicates) by deriving a necessary criterion using the state equation. However, this criterion is often not sufficient, so they use *traps* (a structural feature of Petri nets) to strengthen the conditions.

Structural techniques can also help state space-based approaches. Bønneland et al. [Bøn+18] use the state equation to simplify temporal logic (CTL) formulas. Pastor et al. [PCP99] use place invariants (P-invariants) for more efficient BDD encoding, while Karsten [Sch03] uses both P- and T-invariants to store fewer states and in a more compact way. The LoLa tool [Wol18] uses the Petri net CEGAR approach in its portfolio to answer reachability subqueries.

1.6 Summary and Future Work

In this thesis, we presented various extensions to the CEGAR approach on Petri nets, lifting its expressive power and increasing the amount of conclusive answers. We implemented the original algorithm and our extensions in the PetriDotNet framework and conducted an experimental evaluation. Results show that the new algorithms could outperform existing tools and approaches in terms of conclusive answers or expressive power on various inputs. My contributions are summarized as follows.

> **Thesis 1** I proposed extensions and improvements to the CEGAR-based reachability analysis of Petri nets, lifting its expressive power and increasing the amount of conclusive answers.

> 1.1 I generalized the algorithm to be able to solve reachability of predicates, where the target state to be reached can be described with a set of linear constraints.
> 1.2 I extended the algorithm to be able to handle Petri nets with inhibitor arcs, raising its expressive power.
> 1.3 I defined the concept of distant invariants and proposed a new iteration strategy, which extended the kind of problems the algorithm could solve.
> 1.4 I defined a new ordering between partial solutions and a corresponding hybrid search strategy that can speed up the convergence of the algorithm without losing solutions.

Joint work. András Vörös and Tamás Bartha were taking part in this research as my B.Sc. supervisors. András Vörös gave the proof for inconclusive answers in his Ph.D. thesis [Vör18]. Zoltán Mártonka, a fellow student, developed some optimizations, took part in the implementation, and was responsible for the proof of unsoundness.

Publications. The extensions of inhibitor arcs and predicates were first presented at the SPLST 2013 conference [c4] and later further elaborated in the Acta Cybernetica journal [j1]. The distant invariants were defined in the author's B.Sc. thesis [a20] and then presented at the Petri Nets 2015 conference [c5] along with the hybrid search strategy. The implementation of PetriDotNet (including the plug-in for the algorithms described in this thesis) was presented in a tool paper at the Petri Nets 2016 conference [c7] and elaborated in more detail with applications in the Science of Computer Programming journal [j2].

Applications. The CEGAR algorithm and our new contributions are implemented in PetriDotNet [c7], which is used in education and research projects at the Budapest University of Technology and Economics. The tool and the algorithm were used during an internship at *evopro*[18] for the modeling and analysis of public transportation systems [j2]. Furthermore, PetriDotNet is used in education as a demonstrator tool and for the homework at the Formal Methods course of the Budapest University of Technology and Economics [c7; j2].

Future work. Despite the extensions, there are still problems that our algorithm cannot solve. A significant result would be if either the algorithm could be extended to handle all cases, or the impossibility of making the algorithm always being conclusive could be formally proven. We suspect that a single forward search in the state space (that the algorithm currently does) might not be enough for a conclusive answer in general. For example, the algorithm for deciding the reachability of vector addition systems (equivalent to Petri nets) uses two semi-algorithms: one that tries to prove reachability by enumerating finite sequences and another one that tries to prove non-reachability using Presburger formulas [Ler11]. It would be interesting to extend the CEGAR approach with a similar second component.

A further possible direction could also be to focus on decidable cases by determining subclasses of Petri nets (e.g. by restricting the structure) where the algorithm is provably conclusive. For example, one trivial class is the case of acyclic nets, where the solution to the state equation is not only a necessary but also a sufficient condition for reachability. However, there are various other subclasses that have been studied [EN94], but not in the context of CEGAR.

We only described distant invariants in the context of basic Petri nets. When using inhibitor arcs, new situations arise. For example, a transition might be connected to two places by inhibitor arcs

[18] http://www.evopro.hu/en

where an invariant is just moving a token back and forth between these two places as it does not know that the token must be removed entirely from both of them. Similarly to getting tokens transitively, tokens could also be removed indirectly with the means of distant invariants.

Chapter **2**

Efficient Strategies for CEGAR-based Software Model Checking

This chapter presents our efficient strategies for CEGAR-based software model checking. We start by introducing control-flow automata as a formal model of programs, and a generic CEGAR-based model checking algorithm (Section 2.1). Then, we propose our contributions: a configurable explicit-value analysis domain, an error-guided search strategy, a backward search-based interpolation method, and a refinement approach using multiple counterexamples (Section 2.2). We briefly discuss the implementation of the algorithms in Theta (Section 2.3) and perform an extensive experimental evaluation (Section 2.4). Then, we put our work in context with related literature (Section 2.5). Finally, we summarize the thesis, highlight the contributions, and suggest future directions (Section 2.6).

2.1 Background

This section introduces the preliminaries of this thesis. First, we present propositional and first-order logic along with satisfiability modulo theories (Section 2.1.1). Then, we introduce control-flow automata as the modeling formalism used in our work (Section 2.1.2). Finally we describe the abstraction and CEGAR-based framework (Section 2.1.3), in which we formalize our new algorithms (Section 2.2).

2.1.1 Mathematical Logic

Propositional logic. In propositional logic [BM07], a *formula* is composed of Boolean variables and connectives (such as $\neg, \vee, \wedge$). An *interpretation* I assigns each variable a truth value (true or false). Given a formula φ and an interpretation I we say that $I \models \varphi$ (I "models" φ) if φ evaluates to true under I. A formula φ is *satisfiable* if an interpretation I exists with $I \models \varphi$.

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

SAT was the first problem shown to be NP-complete [Coo71], meaning that no efficient algorithm is believed to exist regarding worst-case complexity. However, modern SAT solvers with careful engineering [Mos+01] can handle problems with up to tens of millions of variables and clauses [Jär+12] by using algorithms like DPLL [DLL62] and CDCL [MS99]. Boolean satisfiability has tremendous importance in computer science, and in formal verification [VWM15] in particular. However, some problems (e.g. reasoning about computer programs) can be expressed more naturally in richer languages.

First-order logic. First-order logic (FOL) [BM07] generalizes and extends propositional logic with predicates, functions, and quantifiers. The satisfiability problem is defined similarly to propositional logic: a formula φ is satisfiable if an interpretation I exists with $I \models \varphi$ [BM07]. Church [Chu36] and Turing [Tur36] proved that satisfiability is undecidable for FOL in the general case. However, in practical applications, the problem is often decidable because there are different background theories, which give particular meaning to predicates and functions, and restrict the signature and the usage of quantifiers.

Satisfiability modulo theories. While satisfiability in FOL is undecidable in general, it is decidable in many practical first-order theories (or their fragments) [KS16]. A *first-order theory* T is defined by (1) a *signature* Σ_T, which is the set of constant, function and predicate symbols, (2) and a set of *axioms* A_T, providing meaning for the formulas [BM07]. A formula φ is satisfiable in T if an interpretation I exists that satisfies the axioms A_T and φ.

> **Definition 2.2 (Satisfiability modulo theories).** The *satisfiability modulo theories* (SMT) problem [BT18; BHM09] is to decide if a formula φ is satisfiable in a theory $T(\Sigma_T, A_T)$.

There are various theories, including equality logic and uninterpreted symbols, linear/nonlinear arithmetic over integers and reals, bitvectors, arrays, pointer logic, and so on [KS16; BFT16]. The decidability and complexity of these theories have a high variance. The interested reader is referred to [BM07] and [KS16] for details. Modern SMT solvers usually work by combining a SAT solver (for the Boolean structure) and theory solvers (to check the consistency of literals provided by the SAT solver) [Seb07].

The example programs in this thesis are usually based on quantifier-free linear arithmetic over integers, but the algorithms presented can work with any theory in general, as long as there is a decision procedure for it. Therefore, in the rest of the chapter, we will simply use FOL formulas in general and assume that an appropriate theory (or their combination [NO79]) exists.

We use the following notations throughout the chapter. Given a set of variables $V = \{v_1, v_2, \ldots\}$ let $V' = \{v'_1, v'_2, \ldots\}$ and $V^{\langle i \rangle} = \{v_1^{\langle i \rangle}, v_2^{\langle i \rangle}, \ldots\}$ represent the *primed* and *indexed* version of the variables. We use V' to refer to successor states and $V^{\langle i \rangle}$ for paths. Given an expression φ over $V \cup V'$, let $\varphi^{\langle i \rangle}$ denote the indexed expression obtained by replacing V and V' with $V^{\langle i \rangle}$ and $V^{\langle i+1 \rangle}$ respectively in φ. For example, $(x < y)^{\langle 2 \rangle} \equiv x^{\langle 2 \rangle} < y^{\langle 2 \rangle}$ and $(x' = x+1)^{\langle 2 \rangle} \equiv x^{\langle 3 \rangle} = x^{\langle 2 \rangle} + 1$. Given an expression φ let $\mathsf{var}(\varphi)$ denote the set of variables appearing in φ, e.g. $\mathsf{var}(x < y + 2) = \{x, y\}$.

2.1.2 Control-Flow Automata

In our work we describe programs using *control-flow automata* (CFA), a graph-based formalism with FOL variables and expressions [BHT07].

> **Definition 2.3 (Control-flow automata).** A control-flow automaton is a tuple $CFA = (V, L, l_0, E)$ where
> - $V = \{v_1, v_2, \ldots, v_n\}$ is a set of variables with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$,
> - L is a set of program *locations* modeling the program counter,
> - $l_0 \in L$ is the *initial* program location,
> - $E \subseteq L \times Ops \times L$ is a set of directed *edges* representing the *operations* that are executed when control flows from the source location to the target.

Operations $op \in Ops$ are either *assignments* or *assumptions* over the variables of the CFA. Assignments have the form $v := \varphi$, where $v \in V$, φ is an expression of type D_v and $\mathsf{var}(\varphi) \subseteq V$. Assumptions have the form $[\psi]$, where ψ is a predicate with $\mathsf{var}(\psi) \subseteq V$. An operation $op \in Ops$ can also be regarded as a transition formula $\mathsf{tran}(op)$ over $V \cup V'$ defining its semantics. For an assignment operation, the transition formula is defined as $\mathsf{tran}(v := \varphi) \equiv v' = \varphi \wedge \bigwedge_{v_i \in V \setminus \{v\}} v_i' = v_i$ and for an assume operation it is $\mathsf{tran}([\psi]) \equiv \psi \wedge \bigwedge_{v \in V} v' = v$. In other words, assignments change a single variable and assumptions check a condition.[1] By abusing the notation, we allow operations $op \in Ops$ to appear as FOL expressions by automatically replacing them with their semantics, i.e. $\mathsf{tran}(op)$.

A *concrete data state* $c \in D_{v_1} \times \ldots \times D_{v_n}$ is a (many sorted) interpretation that assigns a value $c(v) = d \in D_v$ to each variable $v \in V$ of its domain D_v. States with a prime (c') or an index ($c^{\langle i \rangle}$) assign values to V' or $V^{\langle i \rangle}$ respectively. A *concrete state* (l, c) is a pair of a location $l \in L$ and a concrete data state. The set of initial states is $\{(l_0, c) \mid c \in D_{v_1} \times \ldots \times D_{v_n}\}$ and a transition exists between states (l, c) and (l', c') if an edge $(l, op, l') \in E$ exists with $(c, c') \models op$.

A *concrete path* is a finite, alternating sequence of concrete states and operations $\sigma = ((l_1, c_1), op_1, \ldots, op_{n-1}, (l_n, c_n))$ if $(l_i, op_i, l_{i+1}) \in E$ for every $1 \leq i < n$, $l_1 = l_0$, and $(c_1^{\langle 1 \rangle}, c_2^{\langle 2 \rangle}, \ldots, c_n^{\langle n \rangle}) \models \bigwedge_{1 \leq i < n} op_i^{\langle i \rangle}$, i.e. there is a sequence of edges starting from the initial location and the interpretations satisfy the semantics of the operations. A concrete state (l, c) is *reachable* if a path $\sigma = ((l_1, c_1), op_1, \ldots, op_{n-1}, (l_n, c_n))$ exists with $l = l_n$ and $c = c_n$ for some n.

> **Definition 2.4 (Verification task).** A *verification task* is a pair (CFA, l_E) of a CFA and a distinguished error location $l_E \in L$. A verification task is *safe* if (l_E, c) is not reachable for any c, otherwise it is *unsafe* [Bey15].

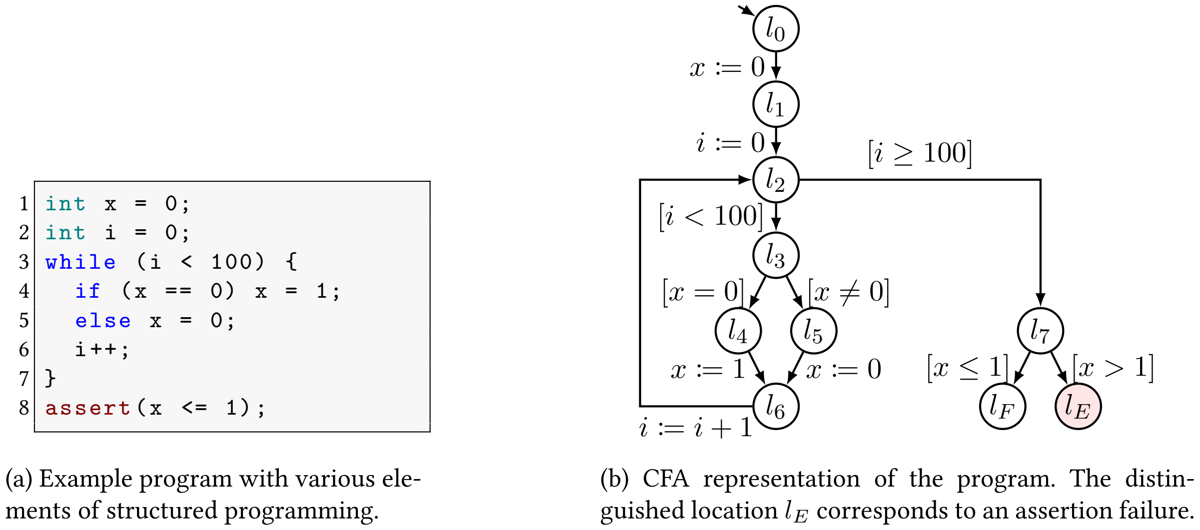
Example. *A simple program and its corresponding CFA can be seen in Figure 2.1. Basic elements of structured programming (sequence, selection, repetition) are represented by the structure of the automaton. The assertion in line* 8 *is mapped as a selection at location* l_7*. If the assertion holds, the program normally ends in the final location* l_F*.*[2] *Otherwise, failure is indicated with the error location* l_E*.*

2.1.3 Counterexample-Guided Abstraction Refinement (CEGAR)

Counterexample-Guided Abstraction Refinement (CEGAR) [Cla+03] is a verification algorithm that automatically constructs and refines abstractions for a given model (Figure 2.2). First, an *abstraction* algorithm computes an *abstract reachability graph* (ARG) [Bey+07] over some abstract domain with respect to a given initial precision. The ARG is an over-approximation of the original state space, and therefore if no abstract state with the error location is reachable, then the original model is also safe [CGL94]. However, if an abstract counterexample (a path to an abstract state with the error location) is found, the *refinement* algorithm checks whether it is feasible in the original model. A feasible counterexample indicates that the original model is unsafe. Otherwise, the counterexample is spurious, the precision is adjusted, and the ARG is pruned so that the same counterexample is not encountered in the next iteration of the abstraction.

[1] Equality constraints do not appear in the implementation, but a single static assignment form [Cyt+91] is used where a new symbol is only introduced when a variable is assigned to.

[2] Note that currently we are not considering termination, i.e. the final location l_F does not carry any special meaning.



(a) Example program with various elements of structured programming.

(b) CFA representation of the program. The distinguished location l_E corresponds to an assertion failure.

Figure 2.1: A simple program and its corresponding CFA, illustrating the correspondence between elements of structured programming (sequence, selection, repetition) and the structure of the CFA.

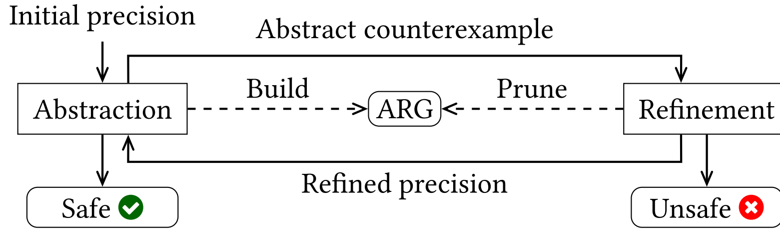


Figure 2.2: Overview of a generic counterexample-guided abstraction refinement (CEGAR) algorithm.

2.1.3.1 Abstraction

We define abstraction based on an *abstract domain* D, a set of *precisions* Π and a *transfer function* T [BHT07].

Definition 2.5 (Abstract domain). An abstract domain is a tuple $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$ where

- S is a (possibly infinite) lattice of abstract states,
- $\top \in S$ is the top element,
- $\bot \in S$ is the bottom element,
- $\sqsubseteq \subseteq S \times S$ is a partial order conforming to the lattice and
- $\mathsf{expr}\colon S \mapsto \mathit{FOL}$ is the expression function that maps an abstract state to its meaning (the concrete data states it represents) using a FOL formula.

By abusing the notation we will allow abstract states $s \in S$ to appear as FOL expressions by automatically replacing them with their meaning, i.e. $\mathsf{expr}(s)$.

Elements $\pi \in \Pi$ in the set of precisions define the current precision of the abstraction. The transfer function $T\colon S \times \mathit{Ops} \times \Pi \mapsto 2^S$ calculates the successors of an abstract state with respect to an operation and a target precision.

In the following, we introduce two domains, namely predicate abstraction and explicit-value abstraction, and their extension with the locations of the CFA.

Predicate abstraction. In *Boolean predicate abstraction* [BPR01; GS97] an abstract state $s \in S$ is a Boolean combination of FOL predicates. The top and bottom elements are $\top \equiv true$ and $\bot \equiv false$ respectively. The partial order corresponds to implication, i.e. $s_1 \sqsubseteq s_2$ if $s_1 \Rightarrow s_2$ for $s_1, s_2 \in S$. The expression function is the identity function as abstract states are formulas themselves, i.e. $\mathsf{expr}(s) = s$.

A precision $\pi \in \Pi$ is a set of FOL predicates that are currently tracked by the algorithm. The result of the transfer function $T(s, op, \pi)$ is the strongest Boolean combination of predicates in the precision that is entailed by the source state s and the operation op. This can be calculated by assigning a fresh propositional variable v_i (also called activation literal) to each predicate $p_i \in \pi$ and enumerating all satisfying assignments of the variables v_i in the formula $s \wedge op \wedge \bigwedge_{p_i \in \pi}(v_i \leftrightarrow p_i')$. For each assignment, a conjunction of predicates is formed by taking predicates p_i with positive variables and their negations $\neg p_i$ with negative variables. The disjunction of all such conjunctions is the successor state s'.

In *Cartesian predicate abstraction* [BPR01] an abstract state $s \in S$ is a conjunction of FOL predicates. Only the transfer function is defined differently than in Boolean predicate abstraction. The transfer function yields the strongest conjunction of predicates from the precision π that is entailed by the source state s and the operation op, i.e. $T(s, op, \pi) = \bigwedge_{p_i \in \pi}\{p_i \mid s \wedge op \Rightarrow p_i'\} \wedge \bigwedge_{p_i \in \pi}\{\neg p_i \mid s \wedge op \Rightarrow \neg p_i'\}$.

Boolean predicate abstraction is more precise, e.g. Cartesian abstraction can only represent $a \wedge \neg b \vee \neg a \wedge b$ as $\top$. However, calculating successors is more efficient in the Cartesian version (linear in the number of predicates) as opposed to Boolean (where it can grow exponentially).

Note that when the precision is empty ($\pi = \emptyset$) the transfer function reduces to a feasibility checking of the formula $s \wedge op$, resulting in *true* or *false* (in both kind of abstractions).

We represent abstract states (in both kinds of abstractions) as FOL formulas. However, a possible optimization would be to use binary decision diagrams (BDDs) for compact representation of states and cheaper partial order checks [Cav+07].

Explicit-value abstraction. In explicit-value abstraction [BL13] an abstract state $s \in S$ is an abstract variable assignment, mapping each variable $v \in V$ to an element from its domain extended with top and bottom values, i.e. $D_v \cup \{\top_{D_v}, \bot_{D_v}\}$. The top element $\top$ with $\top(v) = \top_{D_v}$ holds no specific value for any $v \in V$ (i.e. it represents an unknown value). The bottom element $\bot$ with $\bot(v) = \bot_{D_v}$ means that no assignment is possible for any $v \in V$. The partial order $\sqsubseteq$ is defined as $s_1 \sqsubseteq s_2$ if $s_1(v) = s_2(v)$ or $s_1(v) = \bot_{D_v}$ or $s_2(v) = \top_{D_v}$ for each $v \in V$. The expression function is $\mathsf{expr}(s) \equiv true$ if $s = \top$, $\mathsf{expr}(s) \equiv false$ if $s(v) = \bot_{D_v}$ for any $v \in V$, otherwise $\mathsf{expr}(s) \equiv \bigwedge_{v \in V, s(v) \neq \top_{D_v}} v = s(v)$.

A precision $\pi \in \Pi$ is a subset of the variables $\pi \subseteq V$ that is currently tracked by the analysis. The transfer function is given based on the *strongest post-operator* $\mathsf{sp}\colon S \times Ops \mapsto S$, defining the semantics of operations under abstract variable assignments. Given an abstract variable assignment $s \in S$ and an operation $op \in Ops$, let the abstract variable assignment $\hat{s} = \mathsf{sp}(s, op)$ denote the result of executing op from s. If op is an assumption $[\psi]$ then for all $v \in V$

$$\hat{s}(v) = \begin{cases} \bot_{D_v} & \text{if } s(u) = \bot_{D_u} \text{ for any } u \in V \text{ or } \psi_{/s} \text{ evaluates to } false, \\ s(v) & \text{otherwise,} \end{cases}$$

where $\psi_{/s}$ denotes the expression obtained by substituting all variables in ψ with values from s, except top and bottom values. Note that if ψ is only satisfiable with a single value for a variable v, then the

successor could be made more precise by setting $\hat{s}(v)$ to this value [BL13]. This could be implemented with heuristics[3] for a few simple cases (e.g. $[v = 1]$), but a general solution requires a solver. In our current work, we use a simple heuristic that can detect if an equality constraint has a variable on one side and a literal on the other (e.g. $[v = 1]$) and later we also present a general, configurable solution using a solver in Section 2.2.1. If op is an assignment $w := \varphi$ then for all $v \in V$

$$\hat{s}(v) = \begin{cases} \bot_{D_v} & \text{if } s(u) = \bot_{D_u} \text{ for any } u \in V, \\ s(v) & \text{if } v \neq w, \\ c & \text{if } v = w \text{ and } \varphi_{/s} \text{ evaluates to a literal } c, \\ \top_{D_v} & \text{otherwise.} \end{cases}$$

The transfer function $T(s, op, \pi) = s'$ is defined based on the strongest post-operator sp as follows. Let $\hat{s} = \mathsf{sp}(s, op)$, then $s'(v) = \hat{s}(v)$ if $v \in \pi$ and $s'(v) = \top_{D_v}$ otherwise, for each $v \in V$. In other words, variables not included in the precision are omitted (replaced by top values).

Locations. Locations of the CFA are usually tracked explicitly regardless of the abstract domain used [BHT07]. Given an abstract domain $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$ (e.g. predicate or explicit-value abstraction), let $D_L = (S_L, \bot_L, \sqsubseteq_L, \mathsf{expr}_L)$ denote its extension with locations.[4] Abstract states $S_L = L \times S$ are pairs of a location $l \in L$ and a state $s \in S$. Instead of a single bottom element, there is a set of bottom elements $\bot_L = \{(l, \bot) \mid l \in L\}$ with each location and the bottom element $\bot$ of D. The partial order is defined as $(l_1, s_1) \sqsubseteq_L (l_2, s_2)$ iff $l_1 = l_2$ and $s_1 \sqsubseteq s_2$. The expression function is $\mathsf{expr}_L \equiv \mathsf{expr}$, i.e. the location is not required in the expression as it is encoded in the CFA structure.

Precisions Π are also extended with a location, becoming a function $\Pi_L \colon L \mapsto \Pi$ that maps each location to its precision. Algorithms can be configured to use a *global* precision, which maps each location to the same precision, or a *local* precision, which can map different locations to different precisions.[5]

The extended transfer function $T_L \colon S_L \times \Pi_L \mapsto 2^{S_L}$ is defined as $T_L((l, s), \pi_L) = \{(l', s') \mid (l, op, l') \in E,\ s' \in T(s, op, \pi_L(l'))\}$, i.e. (l', s') is a successor of (l, s) if there is an edge between l and l' with op and s' is a successor of s with respect to the inner transfer function T and the precision assigned to l'.

Abstract reachability graph. We represent the abstract state space using an *abstract reachability graph* (ARG) [Bey+07].

Definition 2.6 (Abstract reachability graph). An abstract reachability graph is a tuple $ARG = (N, E, C)$ where

- $N \subseteq S_L$ is the set of *nodes*, each corresponding to an abstract state in some domain with locations D_L.
- $E \subseteq N \times Ops \times N$ is the set of directed *edges* between locations, labeled with operations. An edge $(l_1, s_1, op, l_2, s_2) \in E$ is present if (l_2, s_2) is a successor of (l_1, s_1) with op.
- $C \subseteq N \times N$ is the set of *covered-by edges*. A covered-by edge $(l_1, s_1, l_2, s_2) \in C$ is present if $(l_1, s_1) \sqsubseteq_L (l_2, s_2)$.

[3] The original paper [BL13] does not exactly mention such heuristics.

[4] Note that technically D_L is not a domain as for example it has no top element. While it is possible to define a generic product domain with locations [BHT07], we rather use locations as a "wrapper" to make our presentation simpler.

[5] In lazy abstraction [Hen+02] the precision can be different even for different instances of the same location.

A node $(l, s) \in N$ is *expanded* if all of its successors are included in the ARG with respect to the transfer function; *covered* if it has an outgoing covered-by edge $(l, s, l', s') \in C$ for some $(l', s') \in N$; and *unsafe* if $l = l_E$. A node that is not expanded, covered, or unsafe is called *unmarked*. An ARG is *unsafe* if there is at least one unsafe node and *complete* if no nodes are unmarked.

An *abstract path* $\sigma = ((l_1, s_1), op_1, (l_2, s_2), op_2, \dots, op_{n-1}, (l_n, s_n))$ is an alternating sequence of abstract states and operations. An abstract path is *feasible* if a corresponding concrete path $((l_1, c_1), op_1, (l_2, c_2), op_2, \dots, op_{n-1}, (l_n, c_n))$ exists, where each c_i is mapped to s_i, i.e. $c_i \models \mathsf{expr}(s_i)$. In practice, this can be decided by querying an SMT solver with the formula[6] $s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge s_2^{\langle 2 \rangle} \wedge op_2^{\langle 2 \rangle} \wedge \dots \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$. A satisfying assignment to this formula corresponds to a concrete path in the CFA.

Abstraction algorithm. Based on the concepts defined above, Algorithm 2.1 presents a basic procedure for abstraction (based on the CPA concept [BHT07]). The input of abstraction is a partially constructed ARG (with possibly unmarked states), an error location l_E, an abstract domain D_L with locations, a current precision π_L and a transfer function T_L. In the first iteration, the ARG only contains the initial state $N_0 = \{(l_0, \top)\}$ and the precision π_L is usually empty, i.e. no predicates or variables are tracked (see the main loop of the algorithm in Section 2.1.3.3).

Algorithm 2.1: Abstraction algorithm.

```
input : ARG = (N, E, C): partially constructed abstract reachability graph
        l_E: error location
        D_L = (S_L, ⊥_L, ⊑_L, expr_L): abstract domain with locations
        π_L: current precision
        T_L: transfer function with locations
output: (safe or unsafe, ARG)
1  waitlist := unmarked nodes from N
2  while waitlist ≠ ∅ do
3      l, s := remove from waitlist
4      // Check if (l, s) is unsafe
5      if l = l_E then
6          return (unsafe, ARG)
7      // Check if (l, s) can be covered
8      else if ∃(l', s') ∈ N : (l, s) ⊑_L (l', s') then
9          C := C ∪ {(l, s, l', s')} // Add covered-by edge
10     // Otherwise (l, s) gets expanded
11     else
12         foreach (l', s') ∈ T_L((l, s), π_L) \ ⊥_L do
13             waitlist := waitlist ∪ {(l', s')}
14             N := N ∪ {(l', s')} // Add new node
15             E := E ∪ {(l, s, op, l', s')} // Add successor edge
16 return (safe, ARG)
```

[6] In software model checking s_1 is usually the top element because the program starts with all variables uninitialized. However, in a more general setting, transition systems can have an arbitrary formula describing the initial states [c6].

The algorithm initializes a waitlist with all unmarked states. Then, it removes and processes states from the waitlist based on some search strategy (e.g. breadth- or depth-first). If the current state corresponds to the error location, the abstraction terminates with an unsafe result and an unsafe ARG. Otherwise, we check if some already reached state covers the current with respect to the partial order. If not, we calculate successors with the transfer function, making the node expanded.

If there are no more nodes to explore and the error location was not found, the abstraction concludes with a safe result and a complete ARG. Note that due to its construction, the ARG without covered-by edges is actually a tree (also called abstract reachability tree (ART) in some works).

Example. *Figure 2.3a shows the ARG for the program in Figure 2.1 using predicate abstraction with a single predicate $\pi_L(l) = \{i < 100\}$ for each location $l \in L$. Nodes are annotated with the location and the predicate (or its negation). Edges are marked with the operations from the CFA. Dashed arrows represent covered-by edges. It can be seen that an abstract state with the error location l_E is reachable, and thus abstraction concludes with an unsafe result. However, using a different set of predicates, e.g. $\pi'_L(l) = \{x \leq 1\}$ it would be able to prove the safety of the program.*

Example. *Figure 2.3b shows the ARG for the same program (Figure 2.1) using explicit-value abstraction with only tracking the variable x, i.e. $\pi_L(l) = \{x\}$ for all $l \in L$. Nodes are annotated with the location and the value of x. It can be seen that no abstract state is reachable in the ARG with the error location l_E, and therefore the original program is safe. Also note that tracking the loop variable i is not necessary, hence reducing the size of the ARG.*

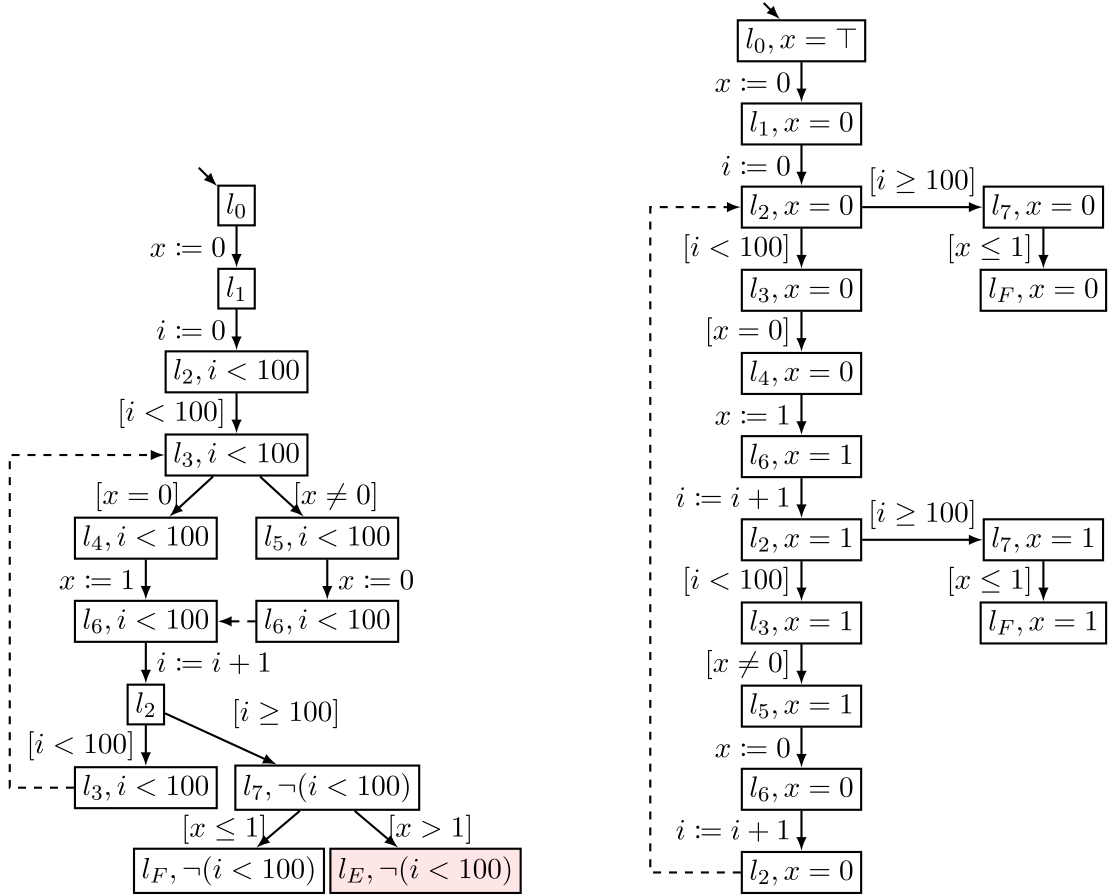
2.1.3.2 Refinement

Refinement checks if a counterexample is feasible in the original program and if not, it adjusts the precision and prunes the ARG so that in the next iteration, this spurious counterexample is eliminated. Algorithm 2.2 presents the refinement procedure. The input is an unsafe ARG, the error location and the current precision π_L.

Feasibility check. Refinement starts with extracting a path $\sigma = ((l_1, s_1), op_1, (l_2, s_2), op_2, \ldots, op_{n-1}, (l_n, s_n))$ to the unsafe state (i.e. $l_n = l_E$) for feasibility checking. A feasible path corresponds to a concrete path (in the original program), leading to the error location, which terminates refinement with an unsafe result. In this case, the precision and the ARG is returned unmodified. Otherwise, an *interpolant* [Cra57; McM05] is calculated from the infeasible path σ that holds information for the further steps of refinement. In our work, we use binary or sequence interpolants.

> **Definition 2.7 (Binary interpolant).** For a pair of inconsistent formulas A and B, an interpolant I is a formula such that
> - A implies I,
> - $I \land B$ is unsatisfiable,
> - $\mathsf{var}(I) \subseteq \mathsf{var}(A) \cap \mathsf{var}(B)$.

A binary interpolant for an infeasible path σ can be calculated by defining $A \equiv s_1^{\langle 1 \rangle} \land op_1^{\langle 1 \rangle} \land \ldots \land op_{i-1}^{\langle i-1 \rangle} \land s_i^{\langle i \rangle}$ and $B \equiv op_i^{\langle i \rangle} \land s_{i+1}^{\langle i+1 \rangle}$, where i corresponds to the longest prefix of σ that is still feasible. Binary interpolants can be generalized to *sequence interpolants* [VG09] in the following way.



(a) ARG for predicate abstraction with precision $\pi_L(l) = \{i < 100\}$ for each $l \in L$. Using this precision the ARG is unsafe because a state with l_E is reachable.

(b) ARG for explicit-value abstraction with precision $\pi_L(l) = \{x\}$ for each $l \in L$. Using this precision, the ARG is safe as no state with l_E is reachable.

Figure 2.3: Example ARGs (with predicates and explicit values) for the program in Figure 2.1. Nodes are represented by rectangles, successors by solid arrows and coverage by dashed arrows.

> **Definition 2.8 (Sequence interpolant).** For a sequence of inconsistent formulas $A_1, \ldots, A_n$, a sequence interpolant $I_0, \ldots, I_n$ is a sequence of formulas such that
> - $I_0 = \mathit{true}$, $I_n = \mathit{false}$,
> - $I_i \wedge A_{i+1}$ implies I_{i+1} for $0 \leq i < n$,
> - $\mathsf{var}(I_i) \subseteq (\mathsf{var}(A_1) \cup \ldots \cup \mathsf{var}(A_i)) \cap (\mathsf{var}(A_{i+1} \cup \ldots \cup \mathsf{var}(A_n)))$ for $1 \leq i < n$.

A sequence interpolant for a path σ can be calculated by defining $A_1 \equiv s_1^{\langle 1 \rangle}$ and $A_i \equiv op_{i-1}^{\langle i-1 \rangle} \wedge s_i^{\langle i \rangle}$ for $1 < i \leq n$. A binary interpolant I_k corresponding to a feasible prefix with length k can also be written as a sequence interpolant where $I_i \equiv \mathit{true}$ for $i < k$, $I_i \equiv I_k$ for $i = k$ and $I_i \equiv \mathit{false}$ for $i > k$. Note that each element I_i of the sequence corresponds to a single state (l_i, s_i) in the counterexample σ, except I_0. Therefore, I_0 is dropped (holds no information), and variables $V^{\langle i \rangle}$ are replaced with V before using the formulas for refinement.

Algorithm 2.2: Refinement algorithm.

input : $ARG = (N, E, C)$: unsafe abstract reachability graph
l_E: error location
π_L: current precision
output: (unsafe or spurious, π'_L, ARG)

1 $\sigma = ((l_1, s_1), op_1, \dots, op_{n-1}, (l_n, s_n)) :=$ path to unsafe node (with l_E) from ARG
2 // Feasibility check
3 **if** $s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge \dots \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$ *is satisfiable* **then return** *(unsafe,* π_L*,* ARG*)*
4 **else**
5 $(I_1, \dots, I_n) :=$ get interpolant for σ
6 // Precision adjustment
7 $(\pi_1, \dots, \pi_n) :=$ map interpolant $(I_1, \dots, I_n)$ to precisions
8 $\pi'_L := \pi_L$
9 **if** π_L *is local* **then** $\pi'_L(l_i) := \pi'_L(l_i) \cup \pi_i$ for each l_i in σ
10 **else** $\pi'_L(l) := \pi'_L(l) \cup \bigcup_{1 \leq i \leq n} \pi_i$ for each $l \in L$
11 // Pruning
12 $i :=$ lowest index for which $I_i \notin \{true, false\}$
13 $N_i :=$ all nodes in the subtree rooted at (l_i, s_i)
14 $N := N \setminus N_i$ // Prune nodes
15 $E := \{(n_1, op, n_2) \in E \mid n_1 \notin N_i \wedge n_2 \notin N_i\}$ // Prune successor edges
16 $C := \{(n_1, n_2) \in C \mid n_1 \notin N_i \wedge n_2 \notin N_i\}$ // Prune covered-by edges
17 **return** *(spurious,* π'_L*,* ARG*)*

Precision adjustment. The precision is adjusted by first mapping the formulas of the interpolant $I_1, I_2, \dots, I_n$ to a sequence of new precisions $\pi_1, \pi_2, \dots, \pi_n$ (in line 7). In predicate abstraction the formulas in the interpolant can simply be used as new predicates, i.e. $\pi_i = I_i$, whereas in the explicit domain variables of these formulas are extracted,[7] i.e. $\pi_i = \mathsf{var}(I_i)$. Then, the new precision π'_L is updated in the following way (in lines 8–10). If π_L is local, then $\pi'_L(l_i)$ is calculated by joining the new precision for each location l_i in the counterexample to its previous precision. Otherwise if π_L is global, then $\pi'_L(l)$ is a union of the old and new precisions for each location $l \in L$.

Pruning. The final step of the refinement is to prune the ARG back until the earliest state where actual refinement occurred, i.e. where the precision changed (lazy abstraction [Hen+02]). Formally, this is the node (l_i, s_i) with lowest index $1 \leq i < n$, for which $I_i \notin \{true, false\}$. Pruning is done by removing the subtree rooted at (l_i, s_i), including all the successor and covered-by edges associated with the nodes of the subtree. Note that during this process, the parent of (l_i, s_i) becomes unmarked (not expanded anymore), and nodes might also get unmarked due to the removal of covered-by edges. Thus, the abstraction algorithm can continue constructing the ARG in the next iteration.

[7] Explicit-value analysis originally performs interpolation with the strongest post operator and constraint sequences [BL13]. We use an SMT-based approach to generalize our algorithms for transition systems [c6], where the transition relation is not limited to assignments and assumptions.

2.1.3.3 CEGAR Loop

Algorithm 2.3 connects the abstraction and refinement methods (described by Algorithm 2.1 and 2.2 respectively) into a CEGAR loop (Figure 2.2). The input of the algorithm is an initial location l_0, an error location l_E, an abstract domain D_L with locations, an initial (usually empty) precision π_{L_0} and a transfer function T_L.

Algorithm 2.3: CEGAR loop.

input : l_0: initial location
l_E: error location
$D_L = (S_L, \bot_L, \sqsubseteq_L, \mathsf{expr}_L)$: abstract domain with locations
π_{L_0}: initial precision
T_L: transfer function with locations
output: safe or unsafe

1 $ARG \coloneqq (N \coloneqq (l_0, \top), E \coloneqq \emptyset, C \coloneqq \emptyset)$
2 $\pi_L \coloneqq \pi_{L_0}$
3 **while** *true* **do**
4 | result, $ARG \coloneqq$ ABSTRACTION$(ARG, l_E, D_L, \pi_L, T_L)$ ➔ Algorithm 2.1
5 | **if** result $=$ *safe* **then return** *safe*
6 | **else**
7 | | result, $\pi_L, ARG \coloneqq$ REFINEMENT(ARG, l_E, π_L) ➔ Algorithm 2.2
8 | | **if** result $=$ *unsafe* **then return** *unsafe*

First, an ARG is initialized with a single node corresponding to the initial location l_0 and the top element of the domain. The current precision π_L is also set to the initial precision π_{L_0}. Then the algorithm iterates between performing abstraction and refinement until abstraction concludes with a safe result or refinement confirms a real counterexample.

2.2 Algorithmic Improvements

In this section, we introduce various improvements related to both the abstraction and the refinement phase of the CEGAR algorithm for software model checking. For abstraction, we define a modified version of the explicit domain where a configurable number of successors can be enumerated (Section 2.2.1). We also propose a new search strategy based on the syntactical distance from the error location (Section 2.2.2). For refinement, we present a novel interpolation strategy based on backward reachability (Section 2.2.3). Furthermore, we introduce a method to use multiple counterexamples for refinement (Section 2.2.4).

2.2.1 Configurable Explicit Domain

Motivation. If an expression cannot be evaluated during successor computation in explicit-value abstraction (e.g. due to top elements in abstract states), it is treated and propagated as the top element (i.e. an arbitrary value) [BL13]. In many cases, this is desirable behavior, which can, for example, avoid explicitly enumerating all possibilities for input variables that can indeed take any value from their domain. However, it is also possible that this behavior prevents successful verification.

Example. *Consider the program in Figure 2.4a. The program is safe, because* $0 < x \wedge x < 5$ *and* $x = 0$ *cannot hold at the same time. However, explicit-value abstraction fails to prove the safety of this program. Even if* x *is tracked by the analysis, its value is unknown (*$x = \top$*) due to the nondeterministic assignment in line 1. The assumption in line 2 is satisfiable, but with multiple values for* x*. Therefore, the algorithm continues to line 3 with* $x = \top$*, where the assumption is again satisfiable (with* $x = 0$*), reaching the assertion violation. At this point, refinement returns the same precision as there are no more variables to be tracked. Thus, the same abstraction is built again, and the algorithm fails to prove safety.*

```
1 int x = nondet();
2 if (0 < x && x < 5) {
3   if (x == 0) {
4     assert(false);
5   }
6 }
```

(a) An example where the top value (in line 2) represents a finite amount of possibilities and thus, could be enumerated.

```
1 int x = nondet();
2 if (x != 0) {
3   if (x == 0) {
4     assert(false);
5   }
6 }
```

(b) An example where the top value (in line 2) represents an infinite amount of possibilities and thus, could not be enumerated.

Figure 2.4: Example programs where safety cannot be proven with explicit-value abstraction due to unknown (top) values.

The problem is that this kind of abstraction can only learn facts like $(0 < x \wedge x < 5)$ by enumerating all possibilities for x. This is actually feasible in this case since there are only 4 different values (successors) for x, and from each of them, the assumption $x = 0$ is unsatisfiable, proving the safety of the program. Similar examples include variables with finite domains (e.g. Booleans) or modulo operations (e.g. $x := y \mod 3$). However, explicitly enumerating all values for a variable is often impractical or even impossible due to the large (or infinite) number of possible values.

Example. *Consider now the program in Figure 2.4b. This program is also safe because* $x \neq 0$ *and* $x = 0$ *cannot hold at the same time. In this case, however, enumerating all values for* x *such that* $x \neq 0$ *is clearly impractical.*[8]

Proposed approach. Motivated by the examples above, we propose an extension of the explicit-value domain [BL13], where in case of a nondeterministic expression, we allow a limited number of successors to be enumerated explicitly. If the predefined limit is exceeded, the algorithm works as previously (treating the result as unknown). This way, we can still avoid state space explosion, but can also solve certain problems that could not be handled previously with traditional explicit-value analysis.

First, we define a modified version of the strongest post-operator (denoted by sp'), which distinguishes unknown evaluation results from top elements (introduced deliberately by the abstraction).

[8] In such cases x should not be tracked explicitly, but rather by predicate abstraction. This can be done statically by analyzing variable roles [DRZ17] or dynamically by using a product abstraction [BHT08; BLW15a]. We have also worked with a B.Sc. student on incorporating a dynamic technique into our framework [e18].

Given an abstract variable assignment $s \in S$ and an operation $op \in Ops$, let the resulting abstract variable assignment $\hat{s} = \mathsf{sp}'(s, op)$ be defined as follows. If op is an assumption $[\psi]$ then for all $v \in V$

$$\hat{s}(v) = \begin{cases} \bot_{D_v} & \text{if } s(u) = \bot_{D_u} \text{ for any } u \in V, \\ \bot_{D_v} & \text{if } \psi_{/s} \text{ evaluates to } \mathit{false}, \\ s(v) & \text{if } \psi_{/s} \text{ evaluates to } \mathit{true}, \\ \text{unknown} & \text{otherwise.} \end{cases}$$

If op is an assignment $w \coloneqq \varphi$ then for all $v \in V$

$$\hat{s}(v) = \begin{cases} \bot_{D_v} & \text{if } s(u) = \bot_{D_u} \text{ for any } u \in V, \\ s(v) & \text{if } v \neq w, \\ c & \text{if } v = w \text{ and } \varphi_{/s} \text{ evaluates to a literal } c, \\ \text{unknown} & \text{otherwise.} \end{cases}$$

The difference between sp and sp' is that if sp' cannot evaluate an assumption or an assignment to a literal then it is treated as a special unknown value. As described in the following, such unknown values can only appear during the intermediate steps of successor computation. Therefore, no special care is needed in the rest of the algorithm, but in principle they could be treated as top values.

Our extended, configurable transfer function $T_k(s, op, \pi)$ works as follows (Algorithm 2.4). It first uses sp' to compute the successor abstract variable assignment of s with respect to op. If an unknown value is encountered, we use an SMT solver to query satisfying assignments of the primed version of variables in π for the expression $s \wedge op$ with the given limit k. This is done with a feedback loop in the following way. We first query a satisfying assignment for the formula $s \wedge op$ and project it to only include variables in π'. Then we add the negation of the assignment as a formula to the solver and repeat this process until the formula becomes unsatisfiable or we exceed k. Note that if there are multiple variables in π', the limit k corresponds to all possible combinations and not to each individual variable separately (which would allow $|\pi'|^k$ total assignments). For example, $\{(x = 1, y = 5), (x = 1, y = 6), (x = 2, y = 6)\}$ counts as 3 assignments, even though both x and y can only take 2 different values.

Algorithm 2.4: Configurable transfer function $T_k(s, op, \pi)$.

input : k: bound for explicitly enumerating successors
s: source state
op: operation
π: target precision
output: $S' \subseteq 2^S$: set of successor states

1. $\hat{s} \coloneqq \mathsf{sp}'(s, op)$
2. **if** *$\hat{s}$ contains any unknown value* **then**
3. | $S' \coloneqq$ query at most k assignments of variables in π' for the formula $s \wedge op$
4. | **if** *more than k assignments are possible* **then** $S' \coloneqq \{\mathsf{sp}(s, op)\}$
5. **else** $S' \coloneqq \{\hat{s}\}$
6. **foreach** $s' \in S'$ **do** $s'(v) \coloneqq \top_{D_v}$ for each $v \in V \setminus \pi$
7. **return** S'

If there are no more than k possible assignments, we treat all of them as new successor states as if they were returned by sp'. Otherwise, if there are more than k assignments, we stop enumerating

them and fall back to using sp instead (which always yields a single successor). Finally, we perform abstraction by setting the non-tracked variables $v \notin \pi$ to top elements in the successors (as it is done in plain explicit-value abstraction). Note that as a special case $k = 1$ is similar to traditional explicit-value analysis because each state has at most one successor. However, if an expression cannot be evaluated (even using heuristics), we use an SMT solver, which makes the analysis more expensive, but also more precise.

Discussion. The advantage of this method is that k can be tuned to reduce the number of unknown values while still avoiding state space explosion. For the example in Figure 2.4a, any k with $k \geq 4$ would work. Currently we experimented with different values for k from a fixed set of values (Section 2.4.2.1). However, it would also be possible to use heuristics for automatically selecting or even dynamically adjusting k during the analysis. Such heuristics could be based on the domain of variables (e.g. Booleans, bounded integers) or the operations (e.g. modulo arithmetic). Furthermore, different k values could be assigned to different locations $l \in L$ in the CFA similarly to a local precision.

Note that since we are enumerating k successors in each step, after n steps, there could be k^n states in the worst case. However, this can only happen if there is a nondeterministic assignment for the variables in each step. Otherwise, we know the exact values of each variable after the first step, and we can evaluate every expression in the subsequent steps in exactly one way.

Operations in the CFA have their corresponding FOL expressions. Therefore, an SMT solver can be used out-of-the-box to enumerate successors. However, our algorithm can work with other strategies (known e.g. from explicit model checkers [Kan+15]) as long as they can enumerate successors for a source state and an operation. Furthermore, since we only need the actual successors if there are no more than k of them, as an optimization, heuristics could be developed that can tell if an expression has more than k satisfying assignments without actually enumerating them.

2.2.2 Error Location-Based Search

Motivation. Recall that the abstract state space can be explored using different search strategies, depending on how the ARG nodes in the waitlist are ordered (Algorithm 2.1). For example, breadth and depth-first search (BFS and DFS) order nodes based on their depth ascending and descending, respectively. These basic strategies, however, use no information from the input verification task.

Proposed approach. To focus the search more effectively, we propose a strategy based on the syntactical distance from the error location in the control-flow automaton. Given a verification task (CFA, l_E) we define the distance $d_E \colon L \mapsto \mathbb{N}$ of each location $l \in L$ to the error location l_E as the length of the shortest directed path from l to l_E without considering the operations. Note that $d_E(l)$ is an under-approximation of the actual distance between l and l_E in the ARG since shorter paths are not possible, but some operations might be infeasible, making the actual (feasible) distance longer. The distances can be calculated (and stored for later queries) at the beginning of the analysis using a backward breadth-first search from the error location.[9] Then from each node (l, s) on the waitlist, we simply remove one where $d_E(l)$ is minimal. However, some examples highlight that loops might trick this approach as well. Therefore, we also experiment with metrics based on a weighted sum of the distance to the error location and the depth of the current node in the ARG.

[9]Locations that are not reachable backward from the error location have a distance of infinity. However, we use a variant [c8] of backward slicing [Wei81] as a preprocessing step to remove such locations.

Example. *Consider the CFA in Figure 2.5a. The distance to the error location l_E is written next to each location. For simplicity, operations are omitted from the edges. Furthermore, suppose that most of the paths are actually feasible at the current level of abstraction, as otherwise, all search strategies perform similarly. It can be seen that the number of paths to the error location scales exponentially with the number of branches (if this diamond-shaped pattern is repeated). Therefore, a traditional BFS approach would cause an exponential execution time. DFS would however, find the first path to l_E quickly for example by exploring $l_0, l_1, l_3, l_4, l_6, l_7, l_E$ in this order. The error location-based approach would act similarly, as it first starts with l_0, discovering its successors l_1 and l_2 both with a distance of 5. Then, by picking, for example, l_1, its only successor is l_3 with a distance of 4. Therefore, the algorithm will pick l_3 (with $d_E(l_3) = 4$) next instead of l_2 (with $d_E(l_2) = 5$), similarly to DFS.*

(a) Example where BFS would need an exponential number of steps to reach l_E.

(b) Example where DFS may unfold the loop l_1, l_2 many times.

Figure 2.5: Examples for error location-based search. Numbers next to the locations denote their distance form l_E without considering operations.

Example. *Consider now the CFA in Figure 2.5b. DFS can easily fail for this case if it is feasible to unfold the loop $l_0, l_1, l_2, l_1, l_2, l_1, l_2 \dots$ many times. However, the error location-based search may also fail if the edge from l_1 to l_6 is not feasible. In this case, the algorithm would also iterate between l_1 and l_2 (as long as possible), since l_3 on the other path has a greater distance.*

Discussion. A possible way to overcome the problem of the above example is to use a combined metric based on the depth of the current node in the ARG (denoted by d_D) and the distance to the error location. However, simply summing the distance and the depth causes each node in Figure 2.5a to become equal in the ordering. Hence, it is reasonable to use a weighted metric $w_D \cdot d_D(s, l) + w_E \cdot d_E(l)$. Assigning a greater weight to d_E can guide the search effectively based on the CFA, while a nonzero weight for w_D can help to avoid unfolding loops too many times. Currently, we experimented with the following five different configurations for the weights (Section 2.4.2.2).

- $(w_E = 0, w_D = 1)$ is a traditional breadth-first search.
- $(w_E = 0, w_D = -1)$ is a traditional depth-first search.
- $(w_E = 1, w_D = 0)$ considers only the distance from the error location.
- $(w_E = 2, w_D = 1)$ combines the distance from the error with the depth (BFS), but with less weight.
- $(w_E = 1, w_D = 2)$ also uses depth and the distance from the error but is biased towards depth.

The first three configurations serve as a baseline, while the last two demonstrate combinations. A possible future work could be to experiment with further values for the weights or with iteration strategies known from abstract interpretation [Bou93].

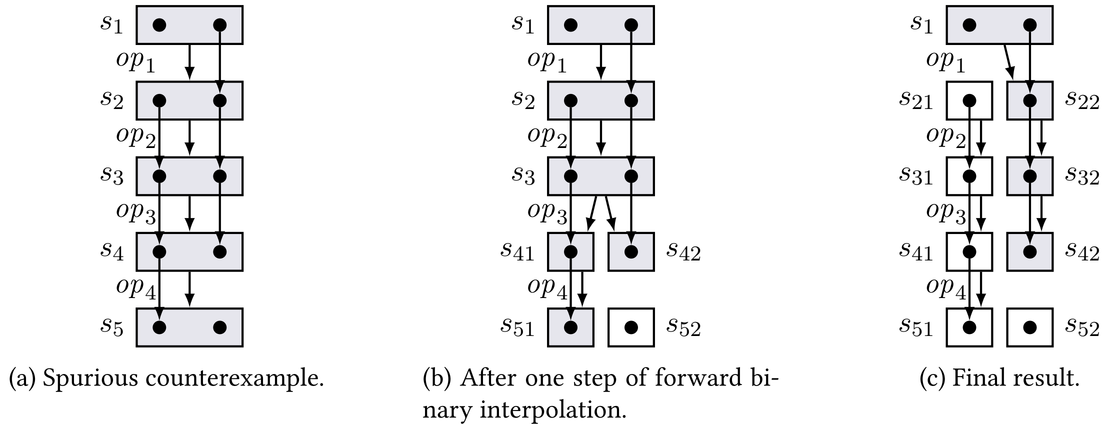
Remark. One might wonder about the usefulness of this approach on safe verification tasks (where no concrete state with l_E is reachable). For such tasks, the intermediate iterations of CEGAR still encounter (spurious) counterexamples. In this case, the error location-based search can help to find these counterexamples and converge towards the final iteration faster.

2.2.3 Backward Binary Interpolation

Motivation. The binary interpolation algorithm presented in Section 2.1.3.2 defines the two formulas A and B based on the longest feasible prefix. This yields an interpolant that refines the last abstract state on the counterexample that can still be reached in the concrete program (starting from the initial state). Therefore, from this point on, we will refer to this strategy as *forward binary interpolation.* We observed that this strategy gives a poor performance in many cases (Section 2.4.2.3).

Example. *Consider the abstract counterexample in Figure 2.6a. Rectangles are abstract states, with dots representing concrete states mapped to them. The initial state is s_1 and the erroneous state is s_5. Edges denote transitions in the concrete and abstract state space. Due to the existential property of abstraction, an abstract transition exists between two abstract states if at least one concrete transition exists between concrete states mapped to them [CGL94].*

It can be seen that the longest feasible prefix is $(s_1, op_1, s_2, op_2, s_3, op_3, s_4)$. Forward binary interpolation would therefore set $A \equiv s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle} \wedge \ldots \wedge op_3^{\langle 3 \rangle} \wedge s_4^{\langle 4 \rangle}$ and $B \equiv op_4^{\langle 4 \rangle} \wedge s_5^{\langle 5 \rangle}$. This gives an interpolant corresponding to s_4, pruning the ARG back until s_3. Continuing from s_3 with the new precision yields s_{41}, s_{42}, s_{51} and s_{52} (instead of s_4 and s_5), as seen in Figure 2.6b. However, s_{51} is still reachable in the abstract state space (via $s_1, s_2, s_3, s_{41}, s_{51}$), but the counterexample is only feasible until s_3 now. The algorithm needs to perform two additional refinements until s_3 and s_2 is refined, and the ARG is pruned back to s_1 (as seen in Figure 2.6c). All spurious behavior is now eliminated as neither s_{51} nor s_{52} is reachable. However, this requires many iterations for the same counterexample and a potentially larger abstract state space in each round due to the increasing precision.



(a) Spurious counterexample. (b) After one step of forward binary interpolation. (c) Final result.

Figure 2.6: Spurious counterexample and its refinement. Rectangles are abstract states, dots represent concrete states mapped to them. A darker background indicates reachable abstract states.

We observed situations as in the example above when a variable is assigned at a certain point of the path (e.g, $op_1 \equiv x := 0$), but only contradicts a guard later (e.g. $op_4 \equiv [x > 5]$). Although the path is feasible until the guard, in these cases, the root cause of the counterexample being spurious traces back to the assignment of the variable.

Proposed approach. To alleviate the previous problems, we define a novel refinement strategy that is based on the longest feasible *suffix* of the counterexample. We call this strategy *backward binary interpolation* as it starts with the erroneous state and progresses backward as long as the suffix is feasible. Formally, let $\sigma = (s_1, op_1, \ldots, op_{n-1}, s_n)$ be an abstract counterexample and let $1 < i \leq n$ be the lowest index for which the suffix $(s_i, op_i, \ldots, op_{n-1}, s_n)$ is feasible. Then we define a backward binary interpolant as $A \equiv s_i^{\langle i \rangle} \wedge op_i^{\langle i \rangle} \wedge \ldots \wedge op_{n-1}^{\langle n-1 \rangle} \wedge s_n^{\langle n \rangle}$ and $B \equiv s_{i-1}^{\langle i-1 \rangle} \wedge op_{i-1}^{\langle i-1 \rangle}$. In other words, A encodes the feasible suffix, and B encodes the preceding transition that makes it infeasible. The formula $A \wedge B$ is unsatisfiable as otherwise, a longer feasible suffix would exist. Similarly to forward binary interpolation, the only common variables in A and B correspond to s_i. Therefore, indexes can be removed from the interpolant I.

Example. *Consider Figure 2.6a again. The longest feasible suffix is* $(s_2, op_2, s_3, op_3, s_4, op_4, s_5)$. *Thus, the interpolation formulas are* $A \equiv s_2^{\langle 2 \rangle} \wedge op_2^{\langle 2 \rangle} \wedge \ldots \wedge op_4^{\langle 4 \rangle} \wedge s_5^{\langle 5 \rangle}$ *and* $B \equiv s_1^{\langle 1 \rangle} \wedge op_1^{\langle 1 \rangle}$. *The resulting interpolant* I *corresponds to* s_2 *and the ARG is pruned back until* s_1 *(Figure 2.6c) in a single step (assuming a global precision).*

Discussion. We motivated backward binary interpolation by comparing it to forward interpolation and showing that it can trace back the root cause in fewer steps. In software model checking, however, sequence interpolation is the standard technique. Hence we also compare our backward interpolation approach to sequence interpolation (Section 2.4.2.3). A potential advantage of backward interpolation is that it can be more compact than sequence interpolation (which could yield a formula for each location along the counterexample, making the algorithm prune a more substantial portion of the state space). Backward search-based strategies also proved themselves efficient in the context of other algorithms, such as Impact [Alb+14] or Newton [Die+17].

2.2.4 Multiple Refinements for a Counterexample

Motivation. Most approaches in the literature stop exploring the abstract state space and apply refinement as soon as the first counterexample is encountered. Although collecting more counterexamples adds overhead to abstraction, better refinements may be possible as more information is available. Altogether, this could reduce the number of iterations and increase the efficiency of the algorithm.

Proposed approach. We modified the abstraction algorithm (Algorithm 2.1) so that it does not return the first counterexample (by removing line 6), but keeps exploring the state space. The algorithm can be configured (by adding a condition to the loop header in line 2) to stop after a given number of erroneous states or to explore all of them.

If at least one of the counterexamples is feasible, then the algorithm can terminate with an unsafe result. However, if all of them are infeasible, there are many possible ways to use the information for refinement. We propose a technique where we first calculate a refinement for each counterexample and derive a minimal set required to eliminate all spurious behavior. Then, we update the precision and apply pruning based on this minimal set.

Our approach is formalized in Algorithm 2.5. First, we extract paths Σ leading to states with the error location l_E from the ARG. If any path $\sigma_i \in \Sigma$ is feasible, then the algorithm terminates with an unsafe result. Otherwise, we calculate an interpolant Itp_i for each path σ_i. Given a path σ_i and its corresponding interpolant Itp_i, we can determine the first state $s_{r_i} \in \sigma_i$ of the path that actually needs refinement (i.e. the first state where the interpolant is not $true$ or $false$). These states correspond to pruning points in the ARG.

Algorithm 2.5: Refinement algorithm for multiple counterexamples.

```
input  : ARG = (N, E, C): unsafe abstract reachability graph
         l_E: error location
         π_L: current precision
output: (unsafe or spurious, π'_L, ARG)
1  Σ = (σ_1, ..., σ_n) := extract paths to states with l_E from ARG
2  // Feasibility check
3  if any path σ_i ∈ Σ is feasible then return (unsafe, π_L, ARG)
4  else
5      π'_L := π_L
6      Itps = (Itp_1, ..., Itp_n) := get interpolant for each σ_i ∈ Σ
7      S_r = (s_{r_1}, ..., s_{r_n}) := calculate first refined state for each σ_i ∈ Σ
8      // Precision adjustment
9      foreach σ_i ∈ Σ do
10         if no state in S_r is a proper ancestor of s_{r_i} in the ARG then
11             (π_{i_1}, ..., π_{i_k}) := map interpolant Itp_i = (I_{i_1}, ..., I_{i_k}) to precisions
12             if π_L is local then π'_L(l_{i_j}) := π'_L(l_{i_j}) ∪ π_{i_j} for each location l_{i_j} in σ_i
13             else π'_L(l) := π'_L(l) ∪ ⋃_{1≤j≤k} π_{i_j} for each l ∈ L
14             // Pruning
15             N_i := all nodes in the subtree rooted at s_{r_i}
16             N := N \ N_i // Prune nodes
17             E := {(n_1, op, n_2) ∈ E | n_1 ∉ N_i ∧ n_2 ∉ N_i} // Prune successor edges
18             C := {(n_1, n_2) ∈ C | n_1 ∉ N_i ∧ n_2 ∉ N_i} // Prune covered-by edges
19     return (spurious, π'_L, ARG)
```

Then, we determine the minimal set of counterexamples to be refined in the following way. For each path σ_i with its first state to be refined s_{r_i}, we check if any other state in S_r is a proper ancestor[10] of s_{r_i} in the ARG. If such a state exists, it means that the other path shares its prefix with the currently examined path, and will need refinement earlier. That refinement will add new predicates and prune the ARG earlier, possibly eliminating the current counterexample as well. Therefore, the current path is skipped for now (lazy refinement).

For each path that is not skipped, we map the interpolant to a new precision and join it to the old one, taking into account whether the precision is local or global. Finally, we return a spurious result, the new precision π'_L, and the pruned ARG.

Example. *Consider the ARG (without covered-by edges) in Figure 2.7. There are four counterexamples $\sigma_1, \ldots, \sigma_4$ in the ARG leading to the abstract states $(s_1, l_E), \ldots, (s_4, l_E)$. The first states to be refined (s_{r_i}) are denoted with a warning sign. In this example the minimal set of counterexamples is $\{\sigma_2, \sigma_3\}$, because s_{r_2} and s_{r_3} are proper ancestors of s_{r_1} and s_{r_4} respectively. Refining σ_2 and σ_3 will, therefore, eliminate all spurious behavior from the current ARG. Note that in the next iteration (s_1, l_E) and (s_4, l_E) might still be reached again if the predicates for σ_2 and σ_3 were not sufficient. In this case, these counterexamples are eliminated in the next iteration.*

[10]Proper ancestors of a node are its ancestors with respect to successor edges, excluding the node itself.

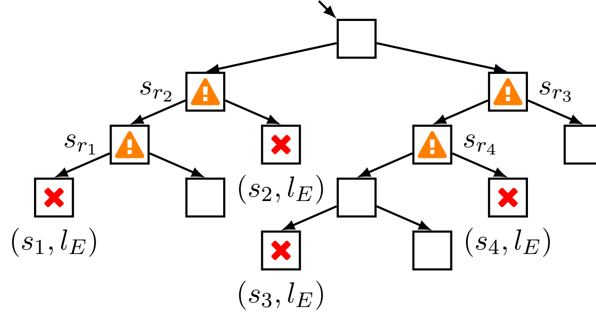


Figure 2.7: Example for refinement based on multiple counterexamples. Counterexamples lead to leaves (s_i, l_E) and the first states to be refined are highlighted by s_{r_i}.

Discussion. Our approach for multiple counterexamples can work with any refinement strategy. In our current experiment (Section 2.4.2.4) we use sequence interpolation. However, it would even be possible to use different strategies for the different counterexamples as opposed to existing approaches that use multiple counterexamples (e.g. DAG interpolation [Alb15] or global refinement [Löw17]).

Currently, we have a single error location in the CFA, so each counterexample leads to the same location on a different path. However, our approach does not rely on this and would work the same way even if the collected counterexamples lead to different locations.

The presented algorithm handles all counterexamples in the solver separately by reusing existing interpolation modules. A possible optimization would be to use the incremental API of SMT solvers by pushing the first counterexample, performing the check and interpolation, and then popping only back to the common prefix of the current and next counterexample, and so on.

2.3 Implementation

We implemented both the existing algorithms presented in the background (Section 2.1) and our new contributions (Section 2.2) in the open source Theta framework[11] [c9]. Theta is a generic, modular and configurable framework, supporting the development and evaluation of CEGAR-based algorithms for the reachability analysis of different formalisms. The main distinguishing feature of Theta is its architecture that allows the definition and combination of different abstract domains, interpreters, and strategies for abstraction and refinement, applied to models of various formalisms with frontends for higher level languages. An overview of the architecture can be seen in Figure 2.8, where the new contributions are marked with a filled background. The architecture follows the modular nature of most modern model checkers [Kor+19]. Theta is written in Java 11.

Formalisms and language frontends. Formalisms are usually based on low-level mathematical representations such as annotated graphs and first-order logic expressions. Formalisms support higher level languages by providing language frontends (consisting of a parser and possibly simplifying reductions). Currently symbolic transition systems (STS) [c6], control-flow automata (CFA, Section 2.1.2) and timed automata [TM17; TM18] are supported with frontends for higher level languages such as AIGER [Bie07; Cab+16], PLC codes [DFB15; Fer+15], C programs [c8] and Uppaal XTA models [LPY97; TM18].

[11] https://github.com/FTSRG/theta (commit f32d3f9 was used for the evaluation)

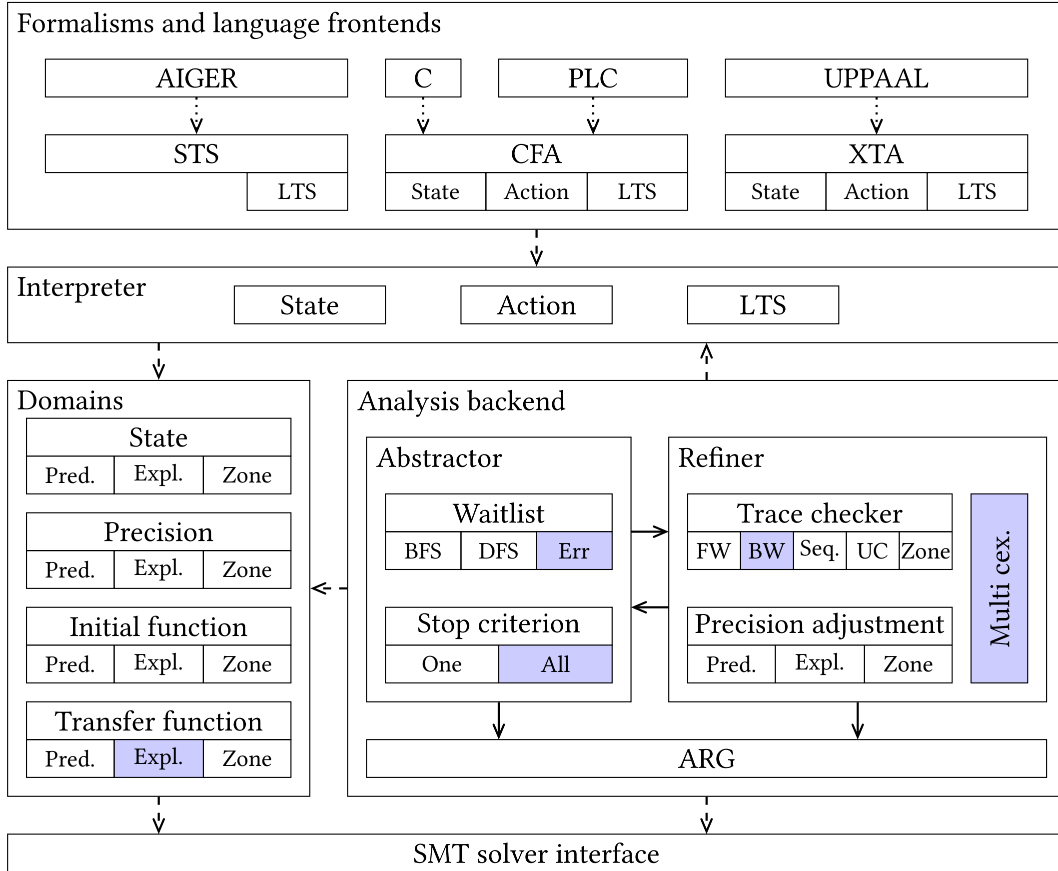


Figure 2.8: Overview of the architecture of Theta. The core of Theta is the analysis backend with the abstract domains. It relies on the SMT solver interface as a backend and on an interpreter and various formalisms and languages as frontends.

Interpreter. The interpreter layer wraps each formalism into a common interface. Each formalism can define its own states, actions, and the labeled transition system (LTS), which determines the enabled actions for a given state. The states of formalisms usually wrap states of abstract domains with extra information. For example, CFA states add a location component to any abstract state, and the CFA LTS returns statements of the outgoing edges as actions.

Domains. The core of the framework is the set of abstract domains with abstract states, precisions, and initial and transfer functions. Currently the predicate, the explicit-value (Section 2.1.3.1) and the zone [Alu99; TM17] domains are implemented. Our configurable explicit domain (Section 2.2.1) is implemented as a variant of the explicit transfer function.

Analysis backend. Reachability analysis is performed by an abstraction refinement loop. As usual for lazy abstraction methods [Hen+02], the central data structure is an abstract reachability graph (ARG). The abstractor component constructs the ARG by expanding and covering nodes. It relies on a waitlist to guide the traversal. Our error location-based search (Section 2.2.2) is implemented as a strategy along BFS and DFS. The abstractor stops when all nodes have been expanded and covered,

or a given amount of unsafe nodes are found. By default, only one unsafe node is required to stop, but our refinement based on multiple counterexamples (Section 2.2.4) explores all unsafe nodes.

The refiner component first checks the trace and extracts the reason of infeasibility. We implemented our novel backward interpolation strategy (Section 2.2.3) along with forward and sequence interpolation (Section 2.1.3.2), unsat cores [LMN15] and zone interpolation [TM17]. The second step of refinement is the precision adjustment, which is done differently for each domain. Our refinement based on multiple counterexamples (Section 2.2.4) is implemented within the refiner by calling trace checking and precision adjustment and then merging the results for all counterexamples.

SMT solver interface. The framework provides a general SMT solver interface that supports incremental solving, unsat cores, and the generation of binary and sequence interpolants. Currently, the interface is implemented by z3 [MB08], but it can easily be extended with new solvers.

2.4 Evaluation

In this section, we evaluate the effectiveness and efficiency of our algorithmic contributions presented before (Section 2.2) by conducting an experiment. First, we introduce our experiment plans along with the research questions to be addressed (Section 2.4.1). Then, we present and discuss our results and analyses for each research question in a separate subsection (Section 2.4.2). Finally, we compare our implementation to other tools in order to provide a baseline for the research questions (Section 2.4.3). The design and terminology of the experiment are based on the book of Wohlin et al. [Woh+12]. The raw data, a detailed report, and instructions to reproduce our experiment are available in a supplementary material [a22].[12]

2.4.1 Experiment Planning

The goal of our experiment is to evaluate the new contributions on a broad set of verification tasks from diverse sources. In our experiment we execute various *configurations* of the CEGAR algorithm on several *input models*.

2.4.1.1 Research Questions

We formulate a research question for the performance of each algorithmic contribution presented in Section 2.2. We are mainly interested in two performance aspects: the number of verification tasks solved within a given time limit per task (effectiveness) and the total execution time required (efficiency). Other measured aspects include a more refined categorization of unsolved tasks (timeout, out-of-memory, exception) and the peak memory consumption.

RQ1 How does the configurable explicit domain perform for increasing values of k compared to traditional explicit-value analysis?

RQ2 How does the error location-based search perform for different weights (w_D, w_E) compared to breadth- and depth-first search?

RQ3 How does backward binary interpolation perform compared to forward binary and sequence interpolation?

RQ4 How does refinement based on multiple counterexamples perform compared to using only a single one?

[12] The supplementary material contains six research questions RQ1, . . ., RQ6. In this dissertation we only present four of them (RQ1, RQ2, RQ4 and RQ5) and they are re-numbered to be RQ1, RQ2, RQ3 and RQ4 respectively.

2.4.1.2 Objects

One of the distinguishing features of Theta is that it supports different kinds of models (e.g. control-flow automata, transition systems, timed automata). An interpreter hides the differences between these formalisms, so the algorithms presented in this thesis work for verification tasks from different domains (e.g. software, hardware). There are some exceptions though: the configurable explicit domain (Section 2.2.1) requires statements and the error location-based search (Section 2.2.2) requires locations. Therefore, these algorithms do not work for hardware models since those are encoded as transition systems.

For the objects of the experiment, we use C programs from the Competition on Software Verification (SV-COMP) [Bey17], hardware models from the Hardware Model Checking Competition (HWMCC) [Cab+16] and industrial programmable logic controller (PLC) software models from CERN [DFB15; Fer+15; DBM19].

Table 2.1 gives an overview of the number of input models and verification tasks along with the size and complexity metrics. We selected models from four categories of the 2018 edition[13] of SV-COMP that are currently supported by the limited[14] C frontend of Theta [c8]. By applying backward slicing [c8], we generate a separate verification task for each assertion. The category *locks* consists of small (94-234 LoC) locking mechanisms with several assertions per model. The collection *loops* includes small (9-70 LoC) programs focusing on loops. The *ECA* (event-condition-action) task set contains larger (591-1669 LoC) event-driven reactive systems. The tasks in *ssh-simplified* describe larger (557-713 LoC) client-server systems.

Table 2.1: Overview of the input verification tasks with the number of variables, locations, edges and the cyclomatic complexity (CC). Ranges denote minimal and maximal numbers.

Source	Category	Models	Tasks	Vars	Locs	Edges	CC
SV-COMP	Locks	13	143	4–32	9–40	10–57	3–23
	Loops	59	105	1–11	4–26	3–33	2–19
	ECA	3	180	9–30	302–1301	375–1516	73–231
	ssh-simpl.	12	17	64–81	187–267	262–375	87–121
CERN	PLC	6	90	1–596	8–4614	7–4782	4–188
HWMCC	HWMCC	300	300	0–245278 inputs, 0–460501 latches, 0–4806245 gates			
	Total	*393*	*835*				

We also experimented with industrial PLC software modules from CERN. These modules operate in an infinite loop, where a formula (the property) is always checked at the end of the loop. The size of these models is greatly varying from a few dozens of locations to a couple of thousands.

Furthermore, we picked all 300 models from the 2017 edition[15] of HWMCC. These tasks are encoded as transition systems, describing circuits with inputs, logical gates, and latches. The metrics reported in the table for the hardware models are after applying the cone of influence (COI) reduction [CGP99].

The majority of the CFA tasks (442) is expected to be safe, while the rest is unsafe (93). To the best of our knowledge, the (300) hardware models do not have an expected result.

[13] https://sv-comp.sosy-lab.org/2018/

[14] Currently Theta does not support arrays, pointers, structs, and function inlining is limited to simple cases.

[15] http://fmv.jku.at/hwmcc17/

Due to slicing [c8], it is possible that different tasks corresponding to the same program will have different models (i.e. CFA). Hence, we encode each task in a separate file, including the model (CFA) and the property, and treat them as if they were different models. Therefore, from now on, we use the terms “model” and “verification task” interchangeably.

2.4.1.3 Variables

The variables of our experiment are listed in Table 2.2, grouped into three main categories. Properties of the model and parameters of the algorithm configuration are independent variables, whereas output metrics of the algorithm are dependent.

Table 2.2: Variables of the experiment.

Category	Name	Type	Description
Model (indep.)	Model	String	Unique name of the model (i.e. verification task).
	Category	Enum.	Category of the model. Possible values: eca, hwmcc, locks, loops, plc, ssh.
Config. (indep.)	Domain	Enum.	Domain of the abstraction. Possible values: EXPL, PRED_BOOL, PRED_CART.
	MaxEnum	Integer	Maximal number of successors to enumerate in the explicit domain (k). Only applicable if Domain is EXPL.
	PrecGranularity	Enum.	Granularity of the precision. Possible values: GLOBAL, LOCAL.
	Refinement	Enum.	Refinement strategy. Possible values: BW_BIN_ITP, FW_BIN_ITP, MULTI_SEQ, SEQ_ITP.
	Search	Enum.	Search strategy. Possible values: BFS, DFS, ERR, DFS_-ERR, ERR_DFS.
Metrics (dep.)	Succ	Boolean	Indicates whether the algorithm successfully provided a correct result within the given resource limits.
	Termination	Enum.	Indicates the termination reason. Possible values: success, time, memory, exception.
	Result	Boolean	Result of the algorithm, indicates whether the model is safe according to the algorithm.
	TimeMs	Integer	CPU time used by the algorithm (in milliseconds).
	Memory	Integer	Peak memory consumption of the algorithm (in bytes).

Properties of the model.

- The variable Model represents the unique name of each model (verification task).
- Furthermore, models have a Category based on their origin.

Parameters of the algorithm.

- The variable Domain represents the abstract domain used. The values PRED_BOOL and PRED_CART stand for Boolean and Cartesian predicate abstraction, while EXPL stands for explicit-value analysis.

- The integer variable MaxEnum corresponds to the maximal number of successors allowed to be enumerated (denoted by k) in our configurable explicit domain (Section 2.2.1). The value 0 represents $k = \infty$, i.e. there is no limit on the number of successors. Furthermore, the value 1^* enumerates at most one successor *without* using an SMT solver[16] (corresponding to traditional explicit-value analysis [BL13]).
- The variable PrecGranularity represents the granularity of the precision. When the granularity is LOCAL, a different precision can be assigned to each location, whereas GLOBAL granularity means that the precision is the same for each location.
- The variable Refinement corresponds to the refinement strategy used. The values FW_BIN_-ITP and SEQ_ITP represent traditional binary and sequence interpolation (Section 2.1.3.2). The value BW_BIN_ITP is our novel backward search-based binary interpolation strategy (Section 2.2.3). The value MULTI_SEQ uses sequence interpolation and our approach of multiple counterexamples (Section 2.2.4).
- The variable Search represents the search strategy in the abstract state space. Values BFS and DFS denote breadth- and depth-first search. Other values correspond to our error location-based strategy (Section 2.2.2) with different weights w_D and w_E. The strategy ERR only takes the error location into account, i.e. $w_D = 0$ and $w_E = 1$. The values ERR_DFS and DFS_ERR use both weights but are biased towards one or the other ($w_D = 2$, $w_E = 1$ and $w_D = 1$, $w_E = 2$ respectively).

Metrics.

- The dependent variable Succ indicates whether the algorithm terminated and provided a correct result (no false negative/positive) successfully within the given CPU time and memory limits (effectiveness).
- The variable Termination indicates the reason for termination (success, timeout, out-of-memory, exception) in a finer way than Succ. It is only used in the detailed plots of the supplementary report [a22].
- The variable Result denotes whether the model is safe or unsafe according to the algorithm. We check that the result matches the expected (if available) and that all configurations agree.
- The variable TimeMs holds the execution time (CPU time) of the algorithm in milliseconds (efficiency).
- The variable Memory measures the peak (maximal) memory consumption during the execution of the algorithm in bytes (efficiency).

2.4.1.4 Experiment Design

The experiment design is summarized in Table 2.3. It would be possible to execute each configuration on every model (crossover design) and then select the relevant subsets of data for each research question. However, due to the high number of parameters and their possible values, it would yield hundreds of configurations. Instead, for each research question, we identify and manipulate one or two parameters that correspond to our new contributions. These parameters are called *factors*, for which each value (level) is executed on every model, and the output is observed.

Based on our previous experience[17] and the literature, the domain of the abstraction is a prominent parameter of CEGAR. Therefore, we also include it in the experiments as a *blocking factor* to

[16] We actually represent 1^* with the integer -1, but keep using 1^* for the clarity of presentation.

[17] Some results of the preliminary experiments were published in [e12; c8; c9; e13]

Table 2.3: Overview of the experiment. Factors and blocking factors are marked with darker and lighter background respectively.

Parameter	RQ1	RQ2	RQ3	RQ4
Domain	EXPL	EXPL, PRED_CART, PRED_BOOL	EXPL, PRED_CART, PRED_BOOL	EXPL, PRED_CART, PRED_BOOL
MaxEnum	0, 1, 1*, 5, 10, 50	plc: 0, sv-comp: 1, hwmcc: NA	plc: 0, sv-comp: 1, hwmcc: NA	plc: 0, sv-comp: 1, hwmcc: NA
Refinement	SEQ_ITP	SEQ_ITP	BW_BIN_ITP, FW_BIN_ITP, SEQ_ITP	MULTI_SEQ, SEQ_ITP
Search	BFS, DFS	BFS, DFS, ERR, DFS_ERR, ERR_DFS	BFS	BFS
PrecGran.	plc: LOCAL, sv-comp: GLOBAL, hwmcc: NA			

systematically eliminate its undesired effect. RQ1 forms an exception, where only the explicit domain is applicable, therefore we use the search strategy for blocking.

The rest of the independent variables are kept at a *fixed level* that usually performed well in our previous experiments. These fixed levels, however, can be different based on the type of the model, e.g. a local precision granularity is used for PLC models, while SV-COMP models perform better with global precision. Furthermore, certain parameters might not be applicable (NA) to hardware models since they are represented as transition systems instead of CFA.

To illustrate our design with an example, in RQ1, we evaluate 6 levels for MaxEnum and 2 levels for Search, while keeping other parameters at a fixed level. This yields a total number of 12 configurations.

2.4.1.5 Measurement Procedure

Measurements were executed on physical machines with 4 core (2.50 GHz) Intel Xeon L5420 CPUs and 32 GB of RAM, running Ubuntu 18.04.1 LTS and Oracle JDK 1.8.0_191. We used z3 version 4.5.0 [MB08] for SMT solving.[18] To ensure reliable and accurate measurements, we used the RunExec tool from the BenchExec suite [BLW19], which is a state-of-the-art benchmarking framework (also used at SV-COMP). Each measurement was executed with a CPU time limit of 300 seconds[19] and a memory limit of 4 GB. The results were collected into CSV files for further analysis. Each measurement was repeated 2 times. Instructions to reproduce our experiment can be found in the supplementary material [a22].

2.4.1.6 Threats to Validity

In this subsection we discuss threats to *construct*, *internal* and *external* validity [Woh+12] of our experiment. We are not concerned with *conclusion* validity, as we do not use statistical tests [Woh+12].

[18] z3 dropped support for interpolation since version 4.8.1, but still works with version 4.5.0 that we used for the measurements. However, in order to use more recent versions, we are considering to use a separate SMT solver for interpolation, e.g. SMTInterpol [CHN12] or MathSAT [Cim+13].

[19] RunExec also puts a limit on the wall time, which is CPU time limit plus 30 seconds by default.

Construct validity can be ensured by using proper metrics to describe the “goodness” of algorithms. We use the number of solved instances for effectiveness and the total execution time and peak memory consumption for efficiency. These metrics are widely used to characterize model checking algorithms [Bey17; Cab+16; Amp+19] and solvers [Jär+12; Web+19].

Internal validity is concerned with identifying the proper relationship between the treatments and the outcome. We use dedicated hardware machines and repeated executions to reduce noise from the environment. Accuracy of the results is ensured by BenchExec [BLW19], a state-of-the-art benchmarking tool. We also apply blocking factors to eliminate undesired effects from known factors systematically. Nevertheless, internal validity could still be improved using a full, crossover design (executing all configurations on all models).

External validity is increased by using models from different and diverse sources, including standard benchmark suites (SV-COMP [Bey17] and HWMCC [Cab+16]) and industrial models [Fer+15]. We compared our new contributions with various state-of-the-art algorithms implemented within the same framework. Furthermore, we also compare our implementation to other tools to provide a baseline (Section 2.4.3). However, external validity would benefit from using additional models (for example, from other categories of SV-COMP) and from comparing related algorithms as well. Describing models with additional variables (e.g. size or complexity) besides their category would also further generalize our results.

2.4.2 Results and Analysis

We present the results and analyses for each research question in a separate subsection. Analyses were performed using the R software environment [Cor17; WG16] version 3.4.3. We only present the most relevant results in this thesis, but the raw data, the R script, and a detailed report can be found in the supplementary material [a22].

In each analysis, we first merge the repeated executions of the same measurement (same configuration on the same model) into a single data point in the following way. We consider a measurement successful if at least one of the repeated executions is successful. This is a reasonable choice as, in most cases, either all executions are successful or none of them are. Then, we calculate the execution time of a measurement by taking the mean time of its successful repetitions. The relative standard deviation[20] between the repeated executions was usually around 1% to 2%, allowing us to represent them with their mean. In a few cases, the repeated executions terminated due to a different reason (e.g. timeout first, then out-of-memory). In these cases, we counted the first reason during aggregation.

2.4.2.1 RQ1: Configurable Explicit Domain

Results. In this question we analyze 6 different levels for MaxEnum with respect to 2 levels for the blocking factor Search. These configurations are only applicable to the 535 CFA models, giving a total number of $(6 \cdot 2) \cdot 535 = 6420$ measurements, from which 3928 (61%) are successful.

The heatmap in Figure 2.9 presents an overview of the results. Configurations are described by the levels of Search and MaxEnum. Categories are given by their name and the number of models, and the rightmost column is a summary of all categories. Each cell represents the number of successful measurements in a given category, along with the total execution time and peak memory consumption for successful measurements (rounded to 3 significant digits [BLW19]). The background color of the cell indicates the success rate of the configurations, i.e. the percentage of successful measurements.

[20]Relative standard deviation (also called the coefficient of variation) is the ratio of the standard deviation to the mean.

The last row is the virtual best configuration, i.e. taking the result of the best configuration for each model individually.

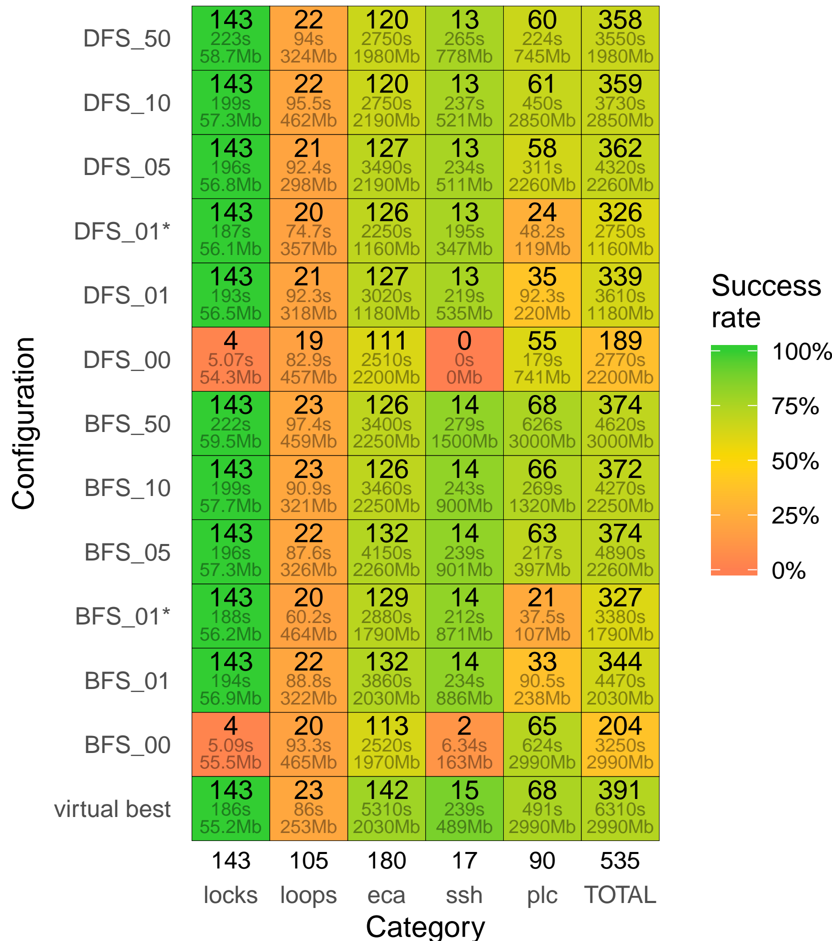


Figure 2.9: Overview of the success rates, total execution time and peak memory usage for RQ1.

Discussion. It can be seen that traditional explicit-value analysis, i.e. configurations BFS_01* and DFS_01* perform well for the SV-COMP categories (locks, eca, ssh), but give poor performance on PLC models. We observed here that the required variables get tracked, but the same counterexample is obtained continuously (there is no refinement progress). One reason for this is that there are nondeterministic Boolean inputs, but the program ensures that not all combinations can be taken (e.g. assuming an implication between two variables allows 3 out of 4 combinations). Without enumeration the analysis cannot represent this and treats them as top values, leading to a spurious counterexample.

On the other end of the spectrum, configurations BFS_0 and DFS_0 enumerate all possible successors ($k = \infty$). This gives a poor success rate on particular SV-COMP categories, having integer variables with a theoretically infinite[21] domain. Lazy abstraction also plays a role in this because we

[21] SV-COMP contains C programs where integers have a fixed bit-width. However, in our implementation, we use SMT integers with an infinite domain. From a practical point of view, enumerating 2^{32} or 2^{64} states can be considered as infinite.

observed that sometimes a variable does not get pruned back to its first assignment (but rather to the first point of infeasibility). Continuing abstraction with enumeration yields infinite successors, because the variable gets tracked, but its value is nondeterministic. Note that these configurations can still solve certain problems as they represent nondeterministic variables with the top value initially and only start enumerating possible values as soon as they appear in some expression (and are tracked explicitly). These configurations are more suitable for PLC models than traditional explicit-value analysis because PLCs usually contain many Boolean input variables, and it is often feasible to enumerate all possibilities to increase precision (which is often needed, as described above).

The advantage of our configurable approach is demonstrated by the configurations having $5, 10$, or 50 for MaxEnum. These configurations give a good performance overall and a remarkably better success rate on category plc compared to traditional explicit-value analysis. Moreover, with $k >= 10$, configurations can solve a few more plc instances than with enumerating all possibilities. It can also be observed, that using an SMT solver for expressions that cannot be evaluated with simple heuristics (01) can improve success rate compared to not using a solver (01*) with 13 and 17 models for DFS and BFS respectively. Furthermore, it can be seen that BFS is consistently more effective than DFS for the same MaxEnum value. The overall best configuration in this analysis is BFS_50, but BFS_05 and BFS_10 closely follows.

An interesting further research direction would be to determine the optimal value for MaxEnum in advance based on the static properties of the input model (e.g. variable usage [Ape+13]) or to adjust it dynamically during analysis.

> **Summary.** *Our configurable explicit domain can combine the advantages of traditional explicit-value analysis and explicit enumeration of successor states, giving a good performance overall in each category. Furthermore, although using an SMT solver requires more time, it increases precision and achieves a slightly higher success rate.*

2.4.2.2 RQ2: Error Location-Based Search

Results. In this question we analyze 5 different levels for Search with respect to 3 levels for the blocking factor Domain. These configurations are only applicable to the 535 CFA models, giving a total number of $(5 \cdot 3) \cdot 535 = 8025$ measurements, from which 6242 (78%) are successful. The heatmap in Figure 2.10 presents an overview of the results. Configurations are described by the levels of Domain and Search.

Discussion. The overall performance of configurations is similar, ranging from 416 to 447 successful measurements for PRED_* and 357 to 389 for EXPL. However, there are some interesting patterns in specific categories. The blocking factor (Domain) is dominant for the loops, ssh and plc categories: configurations with EXPL perform better for ssh and PRED_* is more effective for loops and plc.

The success rates for different search strategies within the same domain are quite similar with a few notable examples. Our purely error location-based strategy (ERR) yields a higher success rate in general compared to others. We observed that this configuration often requires less iterations with significantly smaller ARGs (e.g. 3 iterations with ARG sizes of 284, 532 and 3590 when BFS has 6 iterations with ARGs of size 300, 582, 894, 1522, 2752 and 10766). In contrast, our ERR_DFS combined strategy has a poor performance for eca models in the predicate domain. The supplementary report [a22] includes separate plots for safe and unsafe benchmarks and we observed that the poor performance can be attributed to the safe models. We checked some individual examples with a larger time limit. In this case these configurations also terminated, but required more than 400% time and

150% iterations. We also observed that the advantage of ERR strategies is more prominent for unsafe models, and they are similar to others for safe instances.

A possible future research direction is to experiment with different combinations and weights for the strategies, possibly based on domain knowledge about the input models.

Summary. *Our error location-based search can yield improvement for certain models. However, our combined strategies that are efficient for artificial examples (Figure 2.5) provide no remarkable improvement for real-world models.*

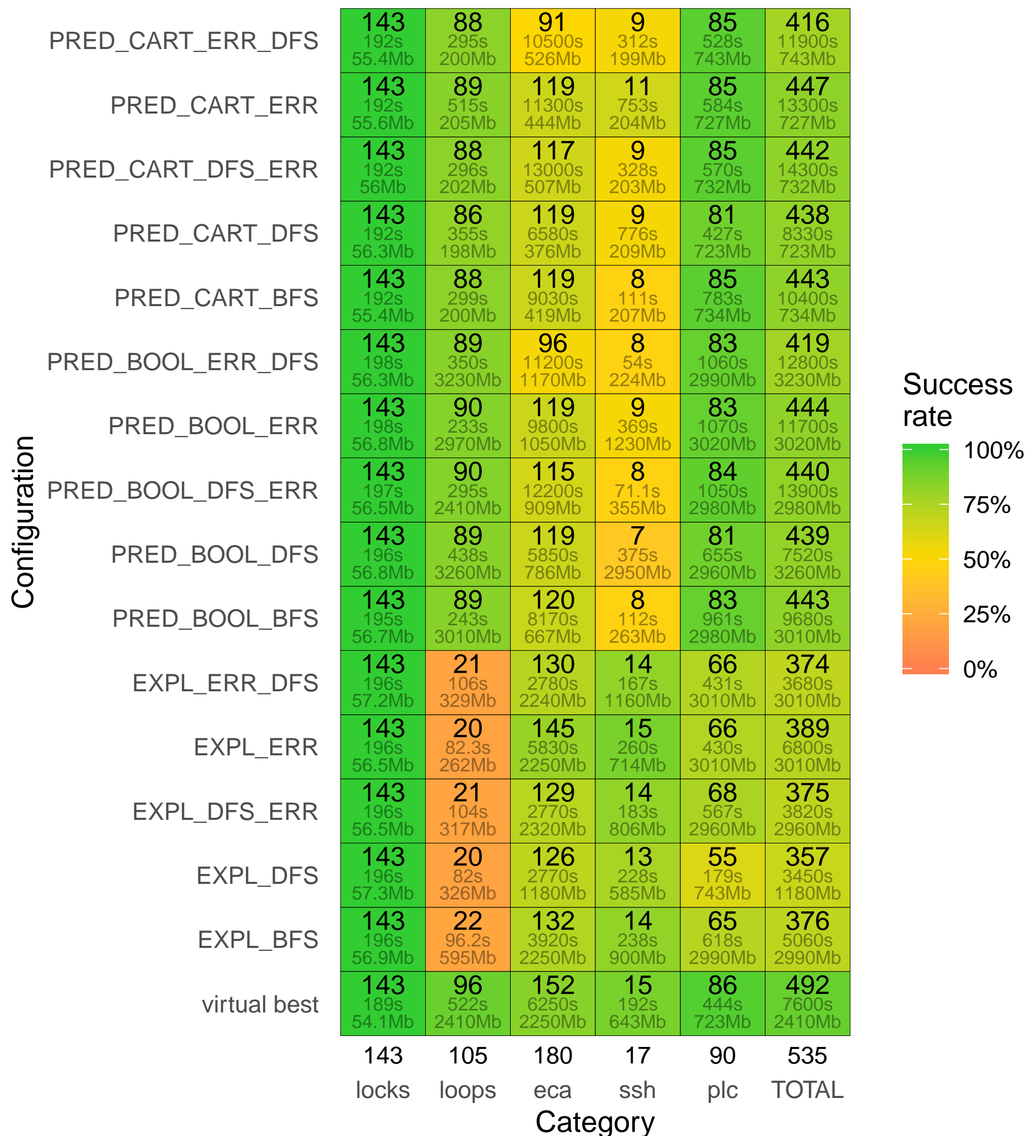


Figure 2.10: Overview of the success rates, total execution time and peak memory usage for RQ2.

2.4.2.3 RQ3: Backward Binary Interpolation

Results. In this question we analyze 3 different levels for Refinement with respect to 3 levels for the blocking factor Domain. These configurations are applicable to all 835 models, giving a total number of $(3 \cdot 3) \cdot 835 = 7515$ measurements, from which 2933 (39%) are successful. The heatmap in Figure 2.11 presents an overview of the results. Configurations are described by the levels of Domain and Refinement.

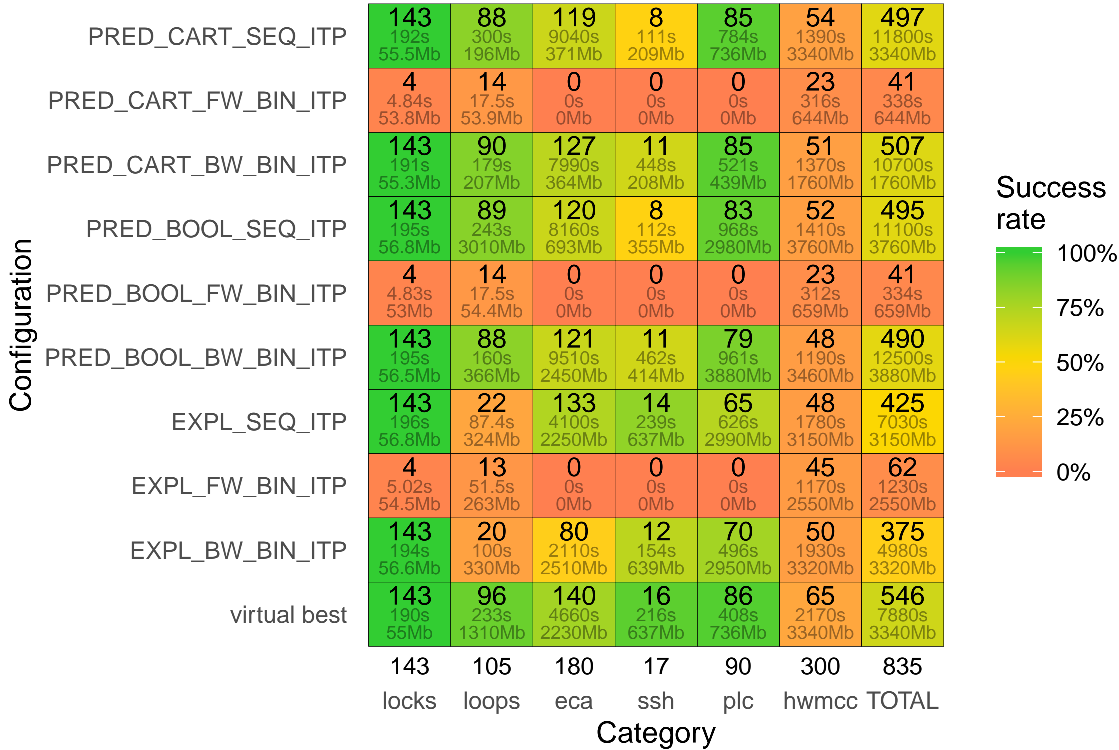


Figure 2.11: Overview of the success rates, total execution time and peak memory usage for RQ3.

Discussion. It can be seen clearly that forward binary interpolation (FW_BIN_ITP) fails for almost every CFA model, except for a few (mainly unsafe) instances in categories locks and loops. For hardware models, it is slightly more effective in the EXPL domain. In most cases we observed that forward interpolation cannot solve problems because late pruning (as discussed in Section 2.2.3): even if the right predicates or variables get tracked they are still unknown before the pruning point and there is no refinement progress in the next iteration.

Sequence interpolation (SEQ_ITP) and our backward binary interpolation approach (BW_BIN_-ITP) have similar success rates. The former one is more successful in the PRED_BOOL and EXPL domains, while the latter is effective in the PRED_CART domain (making it the best overall configuration). The differences are however, only remarkable in the EXPL domain, where BW_BIN_ITP has a low success rate on eca models. Furthermore, BW_BIN_ITP in the PRED_CART domain has around half the peak memory consumption than SEQ_ITP in any domain. We observed that this is due to more compact refinements, which often yield an ARG that is 2–4 times smaller.

An interesting further direction would be to involve the granularity of the precision (local/global) as a blocking factor, as for BW_BIN_ITP a local precision could involve more refinement steps.

> **Summary.** *Our backward binary interpolation strategy clearly outperforms forward interpolation and has similar performance to sequence interpolation, in some cases even outperforming it.*

2.4.2.4 RQ4: Multiple Counterexamples for Refinement

Results. In this question we analyze 2 different levels for Refinement with respect to 3 levels for the blocking factor Domain. We are interested in whether collecting and refining multiple counterexamples at once (MULTI_SEQ) can yield better performance than using a single path (SEQ_ITP). These configurations are applicable to all 835 models, giving a total number of $(2 \cdot 3) \cdot 835 = 5010$ measurements, from which 2858 (57%) are successful. The heatmap in Figure 2.12 presents an overview of the results. Configurations are described by the levels of Domain and Refinement.

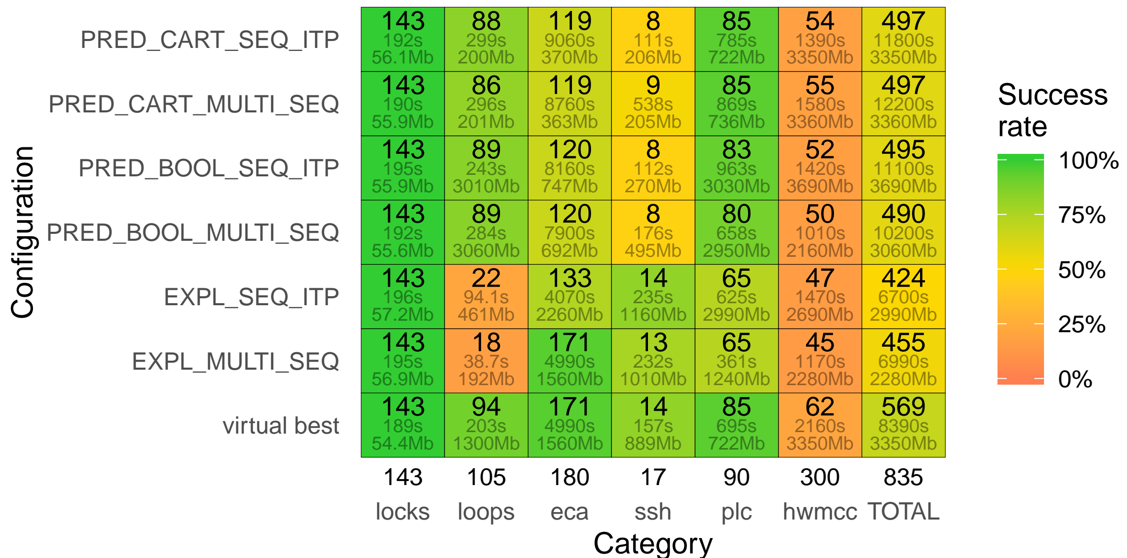


Figure 2.12: Overview of the success rates, total execution time and peak memory usage for RQ4.

Discussion. It can be seen that the blocking factor (Domain) is dominant for the eca, loops, ssh and plc categories. Configurations with PRED_* domain are more successful for loops and plc models, whereas EXPL is more effective for categories eca and ssh. The difference between using a single or multiple counterexamples within the PRED_* domains is not remarkable, ranging from 490 to 497 verified models. However, using multiple counterexamples is clearly more effective in the EXPL domain due to the eca category. This can be attributed to the fact that these models have the largest cyclomatic complexity, enabling the algorithm to utilize the full power of our strategy that uses multiple counterexamples. Furthermore, there are 39 models that only EXPL_MULTI_SEQ could verify. We observed that this can also be attributed to the simpler stop criterion: if we explore all counterexamples, we do not have to check if an unsafe node is found (in each step).

As a possible future direction, it would be interesting to experiment with different refinement strategies (e.g. backward binary). Moreover, information from multiple counterexamples could be utilized in more detail than just selecting a minimal set required to eliminate all spurious behavior.

Summary. *Our strategy for using multiple counterexamples can yield remarkably better performance in the explicit domain for complex models.*

2.4.3 Comparison to Other Tools

In order to provide a baseline for the research questions in the previous section, we compare Theta to other tools. Unfortunately, we did not have the computing resources to run all measurements in a common environment. Therefore, we took the raw data[22] from SV-COMP 2018 and filtered to the models that Theta can handle. Furthermore, we executed four configurations of Theta in a similar environment to SV-COMP, using a 900 second time limit and 15 GB memory limit. Note that these are higher limits compared to the research questions. The hardware machines we used for Theta had weaker CPUs than the ones at SV-COMP, giving us a slight disadvantage. However, our purpose was not to give an exact comparison, but rather just to show that Theta is competitive with respect to the state of the art. Therefore, we omit time and memory measurements and only indicate the number of successful executions.

Currently, the frontend of Theta produces a different verification task for each assertion in a C program due to slicing. Therefore, we selected those models from loops that contain a single assertion. In category locks, there is also a single assertion, reachable by multiple labels that we used as slicing criteria for the research questions. For the current measurements, we create a single task to be able to compare to other tools. The other categories (eca and ssh) contain one assertion per file.

Configurations of Theta are summarized in Table 2.4. The first two configurations (theta-pred-seq and theta-expl-seq) implement already existing strategies, while the latter two include our new approaches that performed well in the research questions. For example, theta-pred-bw performs backward binary interpolation (RQ3), while theta-expl-multiseq uses an SMT solver to evaluate unknown expressions (up to a limit of one) (RQ1) and performs refinement based on multiple counterexamples (RQ4). Furthermore, the latter two configurations use the error location-based search strategy (RQ2).

Table 2.4: Configurations of Theta compared against other tools.

Configuration name	Domain	MaxEnum	Refinement	Search	PrecGran.
theta-pred-seq	PRED_CART		SEQ_ITP	BFS	GLOBAL
theta-expl-seq	EXPL	1*	SEQ_ITP	BFS	GLOBAL
theta-pred-bw	PRED_CART		BW_BIN_ITP	ERR	GLOBAL
theta-expl-multiseq	EXPL	1	MULTI_SEQ	ERR	GLOBAL

Results can be seen in Figure 2.13, where each cell indicates the success rate of a tool (or configuration) in a given category. The last column is a summary of all categories. Empty spaces indicate that a tool did not compete in a certain category. Based on the competition reports [Bey16; Bey17] the tools CPA-BAM-BnB, CPA-BAM-Slicing, CPA-Seq, InterpChecker and Skink are the most closely related to Theta as they also work with CEGAR and ARG-based analysis. The tools UAutomizer,

[22] https://sv-comp.sosy-lab.org/2018/results/results-verified/All-Raw.zip

UKojak and UTaipan also employ CEGAR, but their analysis is based on automata [Bey17]. Other tools are mainly based on bounded model checking, k-induction, or symbolic execution.

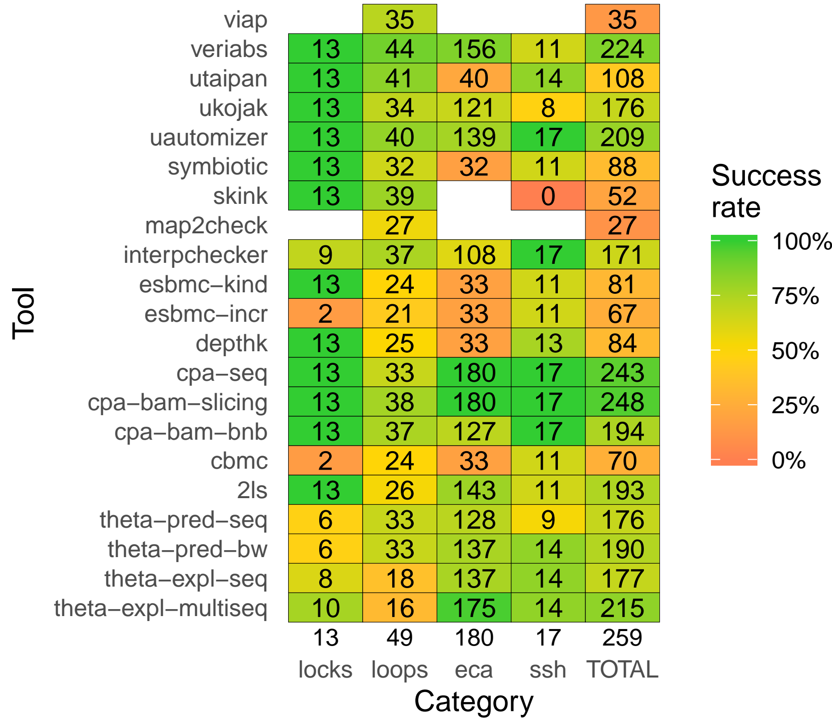


Figure 2.13: Overview of the success rates for various Theta configurations and other tools from SV-COMP 2018.

The models represent a small subset of SV-COMP benchmarks, and many of them belong to the simpler instances. However, the success rates already have a high variance, ranging from 70 to 248 (out of 259) for those tools that competed in all of the categories. Configurations for Theta perform well in this comparison, verifying 176 to 215 tasks. There are 242 tasks (out of 259) that could be solved by at least one Theta configuration. For these models, Figure 2.14 presents their distribution in terms of the number of other tools that could verify them. It can be seen that most of the tasks could be verified by 8 or 9 tools (out of 17), and there are also roughly 20 tasks that only two tools could verify besides Theta. This indicates that there are non-trivial tasks successfully solved by Theta.

Furthermore, Figure 2.15 lists the number of models in each category that were verified by at least one Theta configuration, but not by a given other tool. It can be seen that even though CPA-BAM-Slicing and CPA-Seq can verify more models than Theta configurations in terms of absolute numbers, there are 3 and 7 models respectively in loops that they could not solve. The takeaway message of this comparison is that although the C frontend of Theta is limited, our implementation is still competitive with respect to state-of-the-art tools and can serve as a baseline for evaluating new algorithms and strategies.

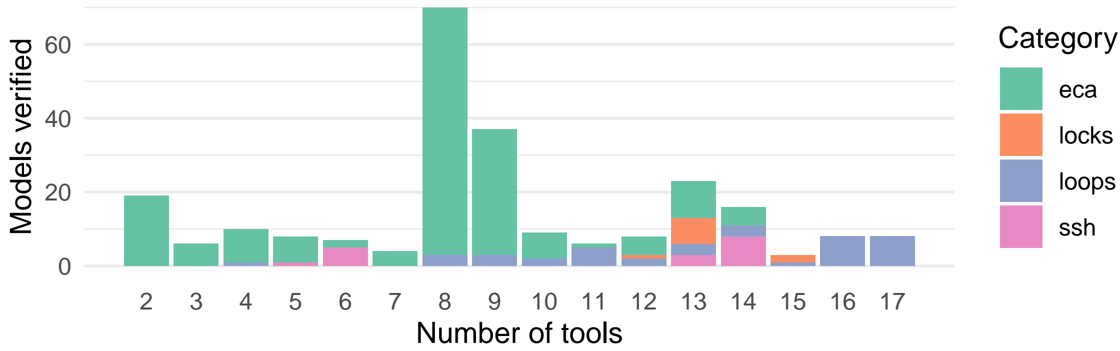


Figure 2.14: Distribution of the 242 models (verified by at least one Theta configuration) in terms of the number of other tools that could verify them.

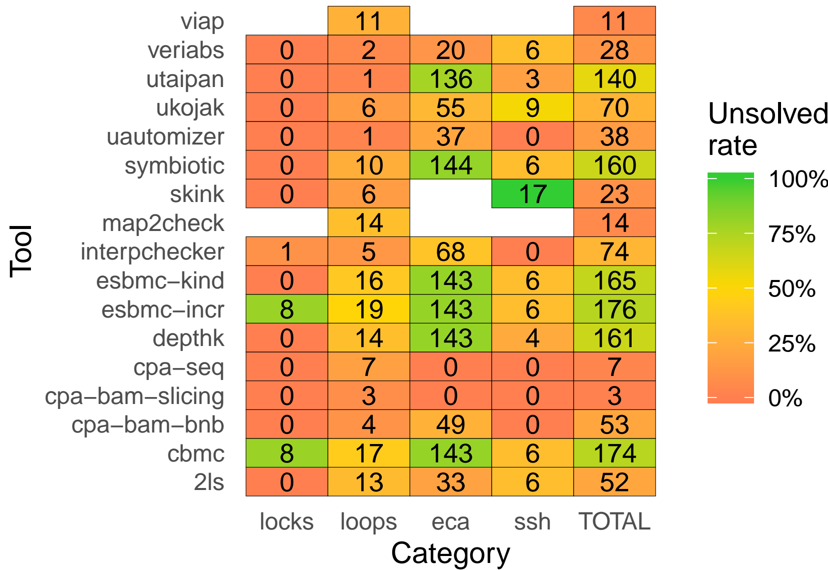


Figure 2.15: Number of models that could be verified by at least one Theta configuration, but not by the given tool.

2.5 Related Work

In this section, we present related work to our framework in general, to our algorithmic contributions and to our experimental evaluation.

General. Abstraction and CEGAR-based methods are widely used for model checking [Bey17], implemented by several tools, e.g. Slam [BR01], Blast [Bey+07], SatAbs [Cla+05], Impact [McM06], Wolverine [KW11], Dagger [Gul+08] and Vinta [AGC12]. The most closely related are the frameworks CPAchecker [BK11] and UFO [Alb+12] that support configurability based on abstract domains and refinement strategies. These tools, however, only target software models in contrast to Theta, which also supports transition systems and timed automata [c9][TM18]. The LTSmin [Kan+15] tool

and the ULTIMATE framework[23] also support different kind of models and algorithms, but their primary focus is on symbolic methods and automata respectively instead of abstraction.

Configurable explicit domain. The transfer function of our configurable explicit domain (Section 2.2.1) can be considered as a generalization of explicit-value analysis [BL13], which always enumerates at most one successor state. The visible/invisible variables approach [CGS04] is similar to the other end of the spectrum, enumerating all possible successors ($k = \infty$) for transition systems defined by partitioned transition relations.

Error location-based search. Our error location-based search (Section 2.2.2) is basically an A* search [HNR68], for which we adjusted the cost function to the domain of software model checking. We use the depth as the cost of the current path, and the distance from the error location as the estimated remaining cost. State space traversal strategies have also been discussed in the context of explicit model checking and abstract interpretation [CC77]. The main focus of these approaches is to reach a fixpoint by identifying widening points and iterations strategies (e.g. based on loops) [Bou93]. In contrast, the goal of our method is to guide the search towards an abstract state with a specific location. However, some ideas of the existing approaches could also be combined with our method, e.g. processing loops first, and then heading towards the error location. Considering the syntactical distance in the CFA has also been proven effective for achieving higher coverage in dynamic test generation tools such as CREST [BS08] and KLEE [CDE08].

Backward binary interpolation. The most closely related to our backward binary interpolation (Section 2.2.3) is the approach of Brückner et al. [Brü+07]. They first calculate a minimal subpath of the counterexample that is spurious, i.e. it is feasible, but extending it in any direction makes it infeasible. Then, they use a binary interpolant to refine the last state of this subpath. In contrast, our approach can be considered as refining the state before the first state of the subpath. Henzinger et al. [Hen+04] also use binary interpolation, but they calculate an interpolant for each location in the counterexample from the same proof. The counterexample minimization approach of Alberti et al. [Alb+14] is also similar to ours as they consider the shortest infeasible suffix of the counterexample. However, their approach is defined in the context of lazy abstraction with interpolants (IMPACT [McM06]), and they compute an interpolant for each location. Moreover, they perform a backward unwinding, whereas we do a forward search and then proceed backward only in the counterexample. This also highlights the possibility to experiment with different combinations of forward/backward search and interpolation.

The NEWTON approach [BR02] performs a forward search, but uses the strongest postcondition operator instead of interpolants. However, counterexamples are generalized with symbolic variables, which could be combined with forward or backward interpolation strategies. A different variant [Die+17] of the NEWTON approach performs a backward search during refinement but uses the weakest precondition operator combined with unsatisfiability cores instead of Craig interpolants. The approach of SLAM [Bal04] also performs a backward check on the counterexample but only up to a bounded depth. Then, they use Craig interpolation at each step to weaken the predicates coming from the weakest preconditions.

Cabodi et al. [CNQ11] compare the iterative application of traditional forward interpolation to sequence interpolation. They come to a similar conclusion as us, namely that while sequence interpolation performs refinement at once, traditional interpolation can sometimes have a better performance due to convergence at shorter depths in their case.

[23] https://ultimate.informatik.uni-freiburg.de/

Multiple counterexamples for refinement. Most algorithms in the literature use a single counterexample for refinement. The UFO tool [Alb+12] includes DAG interpolants [Alb15] that refine all counterexamples at once. Our approach for multiple counterexamples (Section 2.2.4) calculates a separate interpolant for each path and minimizes and merges the results. While computing a DAG interpolant seems more efficient than a series of independent interpolations, our approach could also have various advantages. First, different paths could use different refinement procedures (e.g. backward vs. sequence). Second, it would also be possible to do multiple refinements for each path (e.g. by different interpolation approaches or by multiple prefixes [BLW15b]), take the "best" one and merge it with interpolants from the other counterexamples.

The global refinement algorithm from the thesis of Löwe [Löw17] computes a tree of interpolants using a series of interpolations (by reusing common prefixes). Our approach could also gain performance increase from reusing common prefixes (with the incremental API of solvers). However, our approach has the advantage that each counterexample can use any kind of refinement procedure (e.g. backward interpolation). We believe that this is beneficial in the context of a global precision, where the predicates or variables from the interpolants are merged and used globally.

Experimental evaluation. Many works in the literature focus on experimental evaluation and comparison of model checking algorithms [WBK20; BDW18; BW12; Cze+17; Dem+17]. However, they usually focus on a particular domain (e.g. SV-COMP). Our framework allows us to experiment with models from different domains, including SV-COMP, HWMCC, and PLC codes as well. Furthermore, our experiments compare parameters and configurations of a single algorithm (CEGAR), yielding a finer granularity as opposed to most experiments in the literature, where different tools or different algorithms are compared. This allows us to assess the effectiveness and efficiency of our lower level strategies.

2.6 Summary and Future Work

In this thesis, we presented various new strategies for abstraction and refinement in the context of CEGAR-based software model checking. We implemented a generic CEGAR approach and our new contributions in the Theta verification framework and conducted an experimental evaluation. Results highlight various categories of inputs where the new contributions remarkably improved efficiency. My contributions are summarized as follows.

> **Thesis 2** I proposed various improvements and strategies to CEGAR-based software model checking, increasing the efficiency of the algorithm.
>
> 2.1 I generalized explicit-value analysis to be able to enumerate a predefined, configurable number of successor states, improving its precision, but avoiding state space explosion.
>
> 2.2 I adapted a search strategy to the context of CEGAR that estimates the distance from the erroneous state in the abstract state space based on the structure of the software, efficiently guiding exploration towards counterexamples.
>
> 2.3 I introduced an interpolation strategy based on backward reachability, that traces back the reason of infeasibility to the earliest point in the program, yielding a faster refinement convergence.
>
> 2.4 I described an approach for refinement based on multiple counterexamples, which allows exchanging information between counterexamples and provides better quality refinements.

Joint work. András Vörös and Tamás Tóth were taking part in the development of the generic framework for transition systems as my M.Sc. supervisors. István Majzik, the Ph.D. supervisor of Tamás Tóth, also helped with his advice and feedback. Zoltán Micskei was taking part in the development of the new strategies as my Ph.D. supervisor. The implementation of THETA was a joint work with Tamás Tóth. He was mainly responsible for the core of the framework and the algorithms for timed systems, while I developed the CEGAR algorithm related to transition systems and control-flow automata. The C frontend was developed by a M.Sc. student, Gyula Sallai, whom I co-advised.

Publications. The generic framework for transition systems was defined in the M.Sc. thesis of the author [a21] and published at the FORTE 2016 conference [c6]. Preliminary experiments and evaluations were presented at the Ph.D. Minisymposia at BME [e12; e13]. The improvements to abstraction and refinement were published in the Journal of Automated Reasoning [j3]. The implementation was presented in a tool paper at FMCAD 2017 [c9] and in a paper about the C frontend at VPT 2017 [c8].

Applications. The generic CEGAR framework and the algorithmic improvements are implemented as part of the THETA open-source verification framework [c9]. During a project with CERN we integrated THETA as a backend verifier to the PLCVERIF[24] tool [DBM19]. PLCVERIF works by translating the source code of PLC (programmable logic controller) programs to an intermediate (CFA-like) representation, which can then be mapped to the input language of various model checkers [DFB15]. THETA was successfully integrated with PLCVERIF, and an extensive benchmarking session on 90 input PLC codes confirmed that two configurations of THETA (including our new contributions) could together verify all of them.

Various student theses and works are built on THETA [Czi16; Far16; FB18; Teg18], on the generic CEGAR framework [Sal16; ST17; Baj18; Dob19; MV20] and on our new contributions [Sal19]. Furthermore, THETA is also used in education as a demonstrator in the Critical Architectures Laboratory course, where students develop bounded model checking and CEGAR algorithms.

Future work. We experimented with a predefined set of values for the limit of enumeration in the configurable explicit domain. An interesting future direction would be to determine this number in a preprocessing step based on some static properties of the input model. For example, running an interval-based abstract interpretation could give us hints on the approximate range of variables. Alternatively, this number could also be part of the precision so that it could be dynamically adjusted during refinement.

The error location-based search could be investigated in more detail with different combinations and weights for the search strategies, which could also be determined based on the model, or adjusted dynamically runtime. Furthermore, we could borrow ideas from iteration strategies of abstract interpretation [Bou93], for example, processing loops starting from the innermost and progressing outwards.

The backward binary interpolation strategy works well with a global precision based on our experiments. It would be interesting to see how it performs with local precision. We might need to adjust the algorithm to be effective by, for example refining the precision of all states in the feasible suffix.

Multiple counterexamples were refined based on sequence interpolation. A promising direction would be to try alternative strategies (e.g. backward binary). Furthermore, the implementation could be optimized by only pushing the common prefix of counterexamples to the solver once (utilizing the incremental push/pop feature of solvers).

[24] http://cern.ch/plcverif/

Our algorithms are currently limited to reachability queries. Many properties can be reduced to reachability [Bey15; Sal19], but the expressive power of the algorithm could be lifted by supporting a richer language, e.g. linear- or branching-time temporal logic [Sch02]. A preliminary student work addresses lasso-shaped counterexamples in order to support linear temporal logic (LTL) queries in the CEGAR approach [MV20].

In our implementation, we currently treat sequential statements in the CFA as one block and only calculate abstraction at the end of the block. However, large-block encoding (LBE) [Bey+09] also merges branching statements (if-then-else) using disjunctions. This makes abstraction more precise and more efficient as there are intermediate states where no abstraction is required. Adjustable-block encoding (ABE) [BKW10] generalizes this by allowing the merge points to be configurable. We believe that our algorithms could also benefit from such encodings.

The validity of the experimental evaluation could be increased by using additional models (e.g. further categories of SV-COMP) and by comparing to tools implementing similar strategies (e.g. Ufo [Alb+12]). For this, we are currently working on an LLVM-based frontend [Sal19] to support more language features from C. This frontend already includes various compiler optimizations (such as constant propagation or basic slicing [Wei81]), but we believe that additional techniques borrowed from compilers could make verification more efficient (e.g. loop analyses or advanced slicing [SFB07]).

We are also planning to integrate Theta as a verification backend to the Gamma statechart composition framework [Mol+18]. This would allow our algorithms to work with higher level engineering models, such as statecharts. Backward binary interpolation and multiple counterexamples do not depend on the CFA formalism, while the configurable explicit domain and the error location-based search could be generalized to statecharts. In our prior work, we also experimented with a specialized domain that performs abstraction over the structure of statecharts [c15].

Chapter 3

Modular Specification and Verification of Smart Contracts*

This chapter presents our modular specification and verification approach for Solidity smart contracts. We start by introducing the Solidity language with the underlying Ethereum blockchain and modular verification in general (Section 3.1). Then, we propose our contributions: an adaptation of modular specification constructs to the context of smart contracts, domain-specific annotations, a translation from Solidity to an intermediate verification language, and an efficient bit-precise encoding of arithmetic (Section 3.2). We briefly discuss the implementation of the approach in SOLC-VERIFY (Section 3.3) and perform an experimental evaluation (Section 3.4). Then, we put our work in context with related literature (Section 3.5). Finally, we summarize the thesis, highlight the contributions, and suggest future directions (Section 3.6).

3.1 Background

In this section, we introduce the background of our work. First, we present the main concepts of blockchain-based distributed systems (Section 3.1.1) and we introduce Ethereum, a generic decentralized computing platform (Section 3.1.2) with its programming language, Solidity (Section 3.1.3). Then we describe Boogie, an intermediate verification language serving as our formalism (Section 3.1.4). Finally, we present the verification approach, namely, modular program verification (Section 3.1.5).

3.1.1 Blockchain-Based Systems

A blockchain-based system is a decentralized, peer-to-peer network of nodes that maintain a shared database, called the *ledger* that records *transactions*. The main goal is to achieve the integrity of the ledger without trusting any central authority or node. In other words, each node has its own copy of the ledger locally, but they all agree on its content, i.e. each node has the same view as if there was a single global state.

The key idea of the blockchain is a validation process, called *mining*. Nodes can continuously send transactions to the system. Miner nodes (any node can become a miner) will then take a group of transactions and organize them into a *block*. Miners have to solve a computationally intensive[1]

*The author was also affiliated with SRI International (https://www.sri.com) during the work described in this thesis.

[1]This is called *proof of work*, but alternative protocols (e.g. *proof of stake*) also exist.

puzzle to calculate a valid *hash* for the block. However, after the hash has been found, the other participants can easily check its validity. The first miner to find the next block is rewarded, usually with some amount of *cryptocurrency* (that was freshly issued and/or collected from transaction fees). Other nodes then append the new valid block to their local copy. Each block contains the hash of the previous block (as illustrated in Figure 3.1), and therefore the validity of a hash depends on all the previous blocks transitively. This makes it practically impossible for a node to modify an older block and convince other nodes that it is valid.

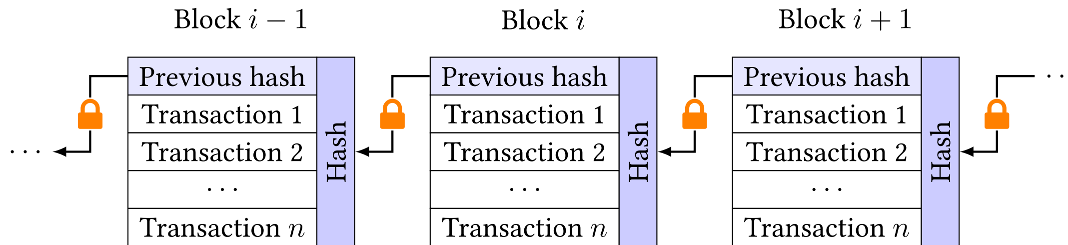


Figure 3.1: Illustration of the blockchain. Transactions are organized into blocks with a hash. Each block transitively depends on all previous blocks by including the hash of the previous block.

An interesting situation arises when different miners validate a new block (with possibly different transactions) nearly at the same time. In this case, a so-called *fork* occurs: one portion of the nodes see one block as the next, while others see the other block. Such situations can be resolved with different consensus protocols (e.g. always picking the longest fork), but it is out of scope for this thesis.

Early applications of the blockchain focused solely on tracking financial transactions of cryptocurrencies (e.g. the Bitcoin [Nak08]). The next step in the evolution of blockchains was to extend the blockchain to a setting where the digital money can also be programmable. This is achieved by generalizing the ledger to allow the deployment of programs (termed *smart contracts* [Sza94]) that operate over ledger data. Blockchains with support for smart contracts provide a general distributed computing platform and allow a set of mutually distrusting parties to execute and enforce their contractual terms (expressed as code) automatically. At the moment, one of the most popular general platforms is the Ethereum blockchain.

3.1.2 Ethereum

Ethereum [Woo17; AW18] is a generic blockchain-based distributed computing platform. The Ethereum ledger is a storage layer for a database of *accounts* (identified by their *address*) and data associated with those accounts. There are two kinds of accounts.

- An *externally owned account* is an account that only has an associated balance and is typically owned by humans (who have the corresponding private key). The balance is stored in terms of *Ether*, the native cryptocurrency of Ethereum.
- A *contract account*, in addition to its own balance, is also associated with the contract state (data) and the compiled contract bytecode that can act on its state.

Ethereum contracts are usually written in a high-level programming language, most notably Solidity [Eth18], and then compiled into the bytecode of the Ethereum Virtual Machine (EVM) [Woo17]. A compiled contract is deployed to the blockchain using a special transaction that carries the contract code and sets up the initial state with the constructor. At that point, the deployed contract is issued an address and stored on the ledger. From then on, the contract is publicly accessible, and its code cannot be modified.

A user (or another contract) can interact with a contract through its public API by calling public functions. This can be done by issuing a *transaction* with the contract's address as the recipient. The transaction contains the function to be called along with the arguments, and an execution fee called *gas*. Optionally, some value of Ether can also be transferred with transactions. The Ethereum network then executes the transaction by running the contract code in the context of the contract instance. Note that this is performed by each miner in parallel, and the result of execution is also verified by each participant after reaching a consensus. Thus, the Ethereum network can be conceptually viewed as a single, global computer.

During their execution, each instruction costs some predefined amount of gas. If the contract overspends its gas limit, or there is a runtime error (e.g. an exception is thrown, or an assertion is triggered), the entire transaction is aborted and has no effect on the ledger (apart from charging the sender for the used gas).

3.1.3 Solidity

The most popular language for writing Ethereum smart contracts to date is *Solidity* [Eth18]. Figure 3.2 shows an example Solidity contract `SimpleBank` that illustrates some of the standard features that Ethereum contracts use in practice. A contract can have *state variables*, which define the persistent data that the contract will store on the ledger. The state of `SimpleBank` consists of a single variable `balances`, which is a *mapping* from addresses to 256-bit unsigned integers. Further Solidity types include *value types*, such as Booleans, signed and unsigned integers (of various bit-lengths), addresses, fixed-size arrays, enumerations, and *reference types*, such as arbitrary-size arrays and structures. Once deployed, an instance of `SimpleBank` will be assigned its address and since no constructor is provided, its data will be initialized to default values (in this case, an empty mapping).

```
1  /// @notice invariant sum(balances) == this.balance
2  contract SimpleBank {
3    mapping(address => uint256) balances;
4
5    function deposit() payable public {
6      balances[msg.sender] += msg.value;
7    }
8
9    function withdraw(uint256 amount) public {
10     require(balances[msg.sender] > amount);
11     if (!msg.sender.call.value(amount)("")) {
12       revert();
13     }
14     balances[msg.sender] -= amount;
15   }
16 }
```

Figure 3.2: An example Solidity smart contract implementing a simple bank. Users can deposit and withdraw Ether with the corresponding functions, and the contract keeps track of user balances. The top level annotation states that the contract will ensure that the sum of individual balances is equal to the total balance in the bank.

Contracts define *functions* that can act on their state. Functions can receive data as arguments, perform computation, manipulate the state variables, interact with other accounts, and finally return some values. In addition to declared parameters, functions also have access to a `msg` structure that

contains the details of the transaction. Our example contract defines two public functions `deposit` and `withdraw`. The `deposit` function is marked as `public` and `payable`, meaning that it can be called by anyone and is allowed to receive Ether as part of the call. This function reads the amount of Ether received from `msg.value` and adds it to the balance of the caller, whose address is available in `msg.sender`. Uninitialized slots in a mapping have the default value of the value type (which is `0` for `uint256`). The received Ether is automatically added to the contract balance in the background.

The `withdraw` function allows users to withdraw a part of their bank balance (as defined by `amount`). The function first checks that the sender's balance in the bank is sufficient using a `require` statement. If the condition of `require` fails, the transaction is reverted with no effect. Otherwise, the function sends the required amount of Ether funds by using the built-in `call` function on the caller address with no arguments (denoted by the empty string). The amount to be transferred with a function call is set with the `value` function. Note that the recipient of the `call` can be another contract. Contracts can define a special *fallback* function that gets executed in such cases and can perform arbitrary actions on its own (within the gas limits) and can also fail (indicating it in the return value of `call`). If `call` fails, the whole transaction is reverted with an explicit `revert`. Otherwise, the balance of the caller is deducted in the mapping as well. Besides `if/else`, Solidity includes standard control structures such as `while`, `do-while`, `for`, `break`, `continue` and `return` with the usual semantics. Further Solidity features are presented briefly along the discussion of the translation to the verification language (Section 3.2.2). The interested reader is referred to the Solidity documentation for more details [Eth18].

Remark. `SimpleBank` contains a classic reentrancy vulnerability that can be exploited to steal funds from the bank. As the control is transferred by `call` to the caller in line 11, before their balance is deducted in line 14, they are free to make another call to `withdraw` to perform a double (or multiple) spending. Although this flaw seems basic, it is the issue that led to the loss of funds in the DAO hack [DMH17].

Example. *As a concrete scenario, consider Figure 3.3 where an instance of the bank is deployed along with an attacker (a) and an honest user (u). The balance of each entity is depicted along their line with a gray background. Furthermore, the content of the mapping inside the bank is included in curly brackets.*

Assume, that initially the bank has 0 Ether, while the user and the attacker have 50 each. First, both the user and the attacker deposits 50 Ether, reducing their balance to 0, and increasing the balance of the bank to 100. The mapping keeps track that now both a and u have 50 Ether in the bank. Now suppose that the attacker calls `withdraw(50)`. The `require` check passes, as `balances[a] >= 50`. This makes the bank call `call()` with a value of 50, reducing its balance to 50 and increasing the balance of the attacker to 50, while also passing over control. In this situation, nothing prevents the attacker from calling `withdraw(50)` again (marked by a warning sign). The problem is that the bank is in an inconsistent state (the mapping still has 50 for the attacker), and the `require` check passes again. This will make the bank call `call()` again with a value of 50 eventually draining its balance and increasing the balance of the attacker to 100. The attacker could repeat this process, but suppose now that it is happy with the outcome and stops. After passing back the control, both calls to `withdraw` finish by deducting the balance of the attacker in the mapping. At the second subtraction, there is actually an underflow (marked by $?$), causing the bank to end up in an inconsistent state.

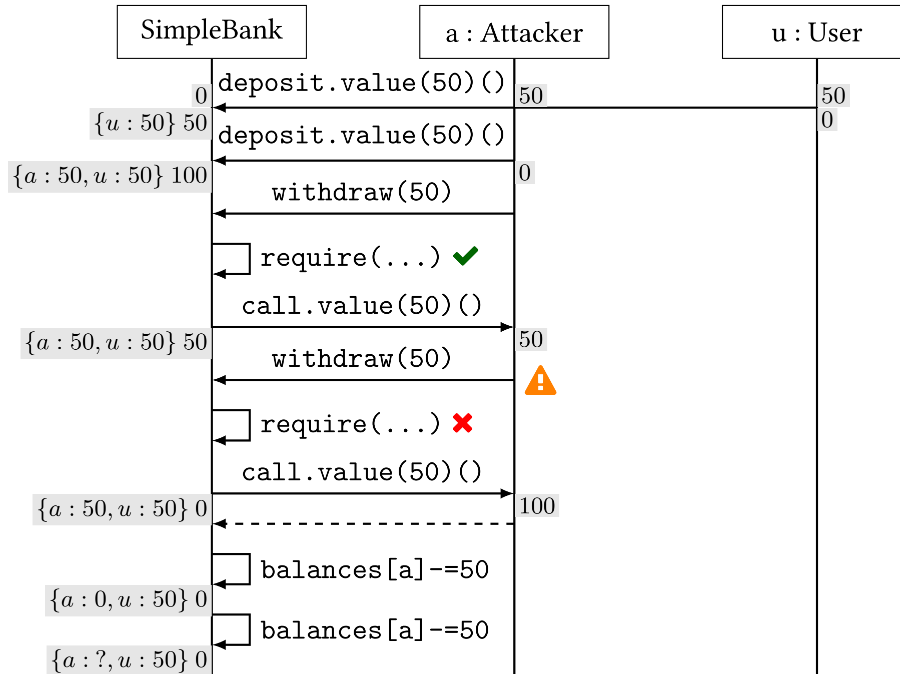


Figure 3.3: A possible reentrancy attack scenario against the bank contract in Figure 3.2. Balances and the content of the internal mapping is displayed along the lines with a gray background.

3.1.4 Boogie IVL

Boogie is an intermediate verification language (IVL) [DL05; Lei08] serving as a layer to build verifiers for various languages. It was originally developed for Spec# [BLS05], but various frontends have emerged since then [Lei10; RE14; MSS16]. We introduce the relevant features of the Boogie IVL briefly. For more details, the reader is referred to [DL05; Lei08] and the documentation.[2]

A Boogie *program* can define *global variables* of various types, including Booleans, mathematical (SMT) integers, bitvectors and SMT arrays [DB09]. Programs can also define *procedures* with *parameters* and (possibly multiple) *return values*. A procedure consists of *local variable declarations* and a block of *statements*.

Statements include assignments, havocs,[3] procedure calls and control structures such as if-then-else, while loop, break, and return. The semantics of such statements are similar to what one would expect from traditional programming languages but are often more restricted. For example, the left hand side (LHS) of an assignment has to be an identifier to a global/local/return variable, and the right hand side (RHS) has to be a side effect free expression (e.g. calls are not allowed). *Expressions* (appearing as part of statements) include literals, identifiers, array read/write, arithmetic, logical operations, and conditionals.

Example. *Figure 3.4a shows a simple Boogie example to illustrate its features. There are two global variables x and y with integer type. The procedure* `add` *takes an integer n, adds n to x and increments y using a loop until it reaches x. The procedure* `incr` *sets x and y to be zero then calls* `add` *with one. There are various specification constructs included in the code, which are introduced next.*

[2] https://boogie-docs.readthedocs.io/

[3] A havoc statement over a variable assigns a nondeterministic value.

```
 1 var x : int;
 2 var y : int;
 3
 4 procedure add(n : int)
 5   requires n >= 0;
 6   requires x == y;
 7   ensures  x == y;
 8 {
 9   x := x + n;
10   while (y < x)
11     invariant y <= x;
12   {
13     y := y + 1;
14   }
15 }
16
17 procedure incr()
18 {
19   x := 0;
20   y := 0;
21   call add(1);
22   assert(x == y);
23   assert(x == 1);
24 }
```

(a) Boogie program illustrating language and specification features.

```
 1 var x : int;
 2 var y : int;
 3 var n : int;
 4
 5 // add(n)
 6   assume(n >= 0);    // requires
 7   assume(x == y);    // requires
 8   x := x + n;
 9   assert(y <= x);    // loop invariant
10   havoc y;
11   assume(y <= x);    // loop invariant
12   goto Body, After;
13 Body:                // start loop body
14   assume(y < x);     // loop condition
15   y := y + 1;
16   assert(y <= x);    // loop invariant
17   goto ;             // end loop body
18 After:               // start after loop
19   assume(!(y < x)); // loop condition
20   assert(x == y);    // ensures
21   goto ;             // end procedure
22
23 // incr()
24   x := 0;
25   y := 0;
26   // call add(1)
27   n := 1;          // set parameter
28   assert(n >= 0); // requires
29   assert(x == y); // requires
30   havoc x;         // modified by add
31   havoc y;         // modified by add
32   assume(x == y); // ensures
33   // end of call add(1)
34   assert(x == y);
35   assert(x == 1); // cannot be proven
36   goto ;           // end procedure
```

(b) Transformation of the program for modular verification where specifications, calls and loops have been replaced.

Figure 3.4: A simple Boogie IVL program and its transformation for modular verification.

Specification. The Boogie IVL includes various constructs for *specification*. Procedures can be annotated with *pre- and postconditions* that must hold respectively before and after the procedure is called. Loops can be annotated with *loop invariants* that must hold before and after every iteration of the loop (including the entry and exit). Furthermore, *assert* and *assume* statements can be placed in procedures. Conditions in asserts must always hold and are checked by verifiers. In contrast, conditions of assumes will not be checked, but just used as being true.

Example. *Recall the Boogie program in Figure 3.4a. Procedure `add` requires its parameter n to be non-negative and global variables x and y to be equal. This holds when `add` is called within `incr`. The loop invariant in `add` states that y should not be greater than x before and after every iteration, which holds due to the preconditions and the increment step of one. In the end, `add` ensures that x and y are equal. Finally, `incr` has two assertions, which also hold after calling `add`.*

3.1.5 Modular Verification

Modular program verification [Poe97; Mül02] is a technique that enables efficient reasoning for composite programs built up from smaller modules, such as classes, interfaces, objects, and procedures. This is achieved by checking each module independently (whether it satisfies its specification) by only relying on the specification of related modules.

Boogie [Bar+06] is a modular verification tool for the Boogie IVL, which treats procedures as the basic unit of verification.[4] Boogie takes programs written in the Booge IVL and translates each procedure into a verification condition (VC) [BL05], which can be discharged by SMT solvers [BT18; BHM09]. The verification condition is essentially an SMT formula, which is valid[5] if and only if the procedure satisfies its specification. We discuss the fundamental ideas of VC generation briefly, which are relevant for the properties (and precision) of modular verification. For more details, the reader is referred to [BL05] and [Lou+15].

As mentioned previously, each procedure is verified independently. Preconditions are assumed at the beginning of the procedure, and postconditions are asserted at the end. Furthermore, calls to other procedures are replaced by their specification at the call site: preconditions are asserted, all variables modified by the callee are havoced, and the postconditions are assumed [LSS99]. Loops are also replaced by their specification (the loop invariant) in the following way [BL05]. The key idea is to first assert the loop invariant (because it must hold before the first iteration). Then, a nondeterministic choice is made between entering into the loop or jumping afterward. In both cases, the variables modified by the loop are havoced (to represent arbitrary iterations), and then the invariant is assumed. If we enter the loop, we assume the loop condition, include the statements of the body, and finally assert the invariant. If we jump after the loop, we assume the negation of the loop condition and continue with the program.

Finally, statements can be translated into an SMT formula by introducing a dynamic single assignment [Fea91]. This is done by introducing a fresh *incarnation* [BL05] of the variable after every update (assign or havoc). Each time a variable is read, its latest incarnation is used. For example, `x := 1; x := x + 1;` becomes `x`$_0$ `:= 1; x`$_1$ `:= x`$_0$ `+ 1;`. This can be trivially done for programs with sequential statements only. If there is branching (e.g. goto), some extra incarnations are needed for unification. For example, if two branches both update x_0 to become x_1 and x_2 respectively, we introduce x_3 `:=` x_1 and x_3 `:=` x_2 respectively to the blocks and use x_3 as the latest incarnation afterwards.

Example. *Recall the Boogie program in Figure 3.4a. Figure 3.4b shows the program after specifications, calls and loops have been replaced (but before dynamic single assignment). The procedure `add` satisfies its specification and can be proven using modular verification. However, asserting `x == 1` in `incr` cannot be proved: the call to `add(1)` will havoc `x` and `y`, and only assume their equality afterwards. Proving `x == 1` would require a stronger postcondition for `add`, namely that it ensures `x == old(x) + n` (where `old` is a special Boogie expression to refer to the value of a variable at the beginning of the procedure).*

3.2 Modular Specification and Verification for Solidity

In this section, we present our contributions, namely a modular specification and verification approach for Solidity smart contracts. Modular verification is a promising direction due to the transactional behavior of the blockchain. We see Boogie as a suitable representation for Solidity contracts as

[4] Later a bounded model checker called Corral [LQL12] was also developed.

[5] A formula is valid if it always evaluates to true. This is equivalent to its negation being unsatisfiable.

most of the language elements (e.g. functions, loops) naturally carry over. Furthermore, the extensive specification possibilities (e.g. pre- and postconditions, invariants) enable a wide range of properties to be expressed and proved automatically with modern SMT solvers. We adapt existing specification annotations to the context of smart contracts but also propose some domain specific extensions (Section 3.2.1). Then, we discuss the translation of contracts (with annotations) to the Boogie IVL (Section 3.2.2), including a scalable bit-precise encoding of arithmetic (Section 3.2.2.7).

3.2.1 Specification Annotations

Solidity provides only a few error handling constructs (e.g. `assert`, `require`, `revert`) for the programmer to specify expected behavior. Therefore, we propose various in-code *annotations* to specify contract properties. With the exception of domain-specific extensions, these annotations follow Solidity expression syntax and typing, making it easy for developers to write and understand the specification. A simple example contract in Figure 3.5 illustrates the different annotations, which we discuss in more detail in the rest of this section.

```
1  /// @notice invariant x == y
2  contract C {
3    int x;
4    int y;
5
6    /// @notice precondition x == y
7    /// @notice postcondition x == (y + n)
8    function add_to_x(int n) internal {
9      x = x + n;
10     require(x >= y); // Catch overflow
11   }
12
13   function add(int n) public {
14     require(n >= 0);
15     add_to_x(n);
16     /// @notice invariant y <= x
17     while (y < x) { y = y + 1; }
18   }
19 }
```

Figure 3.5: An example Solidity smart contract illustrating the annotation features of our approach, including contract-level invariants, pre- and postconditions and loop invariants.

3.2.1.1 Contract Invariants

Similar to object and class invariants [Fla+02; Bar+04; LM05], a *contract invariant* is a constraint over the state variables of the contract that expresses the consistency of the contract state. These constraints must hold at any point after the contract has been deployed and can be called. In order to ensure this, a contract invariant must hold (1) after the contract constructor, (2) after any public function, and (3) before any call to external contracts (as they might call back). In our approach, a contract invariant can be any side-effect free Boolean expression having the same scope as the contract in question (e.g. state variables and `this.balance` can be referenced). Contract invariants are written with specific top-level annotations. An example contract invariant is shown in line 1 of Figure 3.5. During verification, each contract-level invariant is checked (1) as a postcondition to the

constructor, (2) as pre- and postconditions to every public function, and (3) as an assertion before every external call. Contract invariants can also be viewed as an inductive property: they must hold initially (after the constructor), and assuming that they hold before a call, they must also hold afterward.

3.2.1.2 Function Specification with Pre- and Postconditions

As noted above, contract invariants are verified as pre- and postconditions of *public* contract functions. Additional function-specific annotations can be specified per function. Function *preconditions* are Boolean expressions operating over the contract state and function parameters, while *postconditions* can additionally include the return value of the function. This allows full specification of private functions and additional specification of postconditions of public functions.

Annotation of *private* (or internal) functions can be valuable in several respects. First, during modular verification functions are usually substituted with their specification. Therefore, if a private function is not specified, it is assumed that it can perform arbitrary changes to the state, making verification impossible. This can be overcome to some extent by function inlining, and we do perform inlining of private functions to a depth of one by default. Secondly, annotations provide a more gas-friendly alternative to the `require` and `assert` statements in the following way. An equivalent way of ensuring function correctness would be to put the preconditions as `requires` at the beginning of the function and having postconditions as `asserts` at the end of the function. However, these statements are compiled to EVM checks that run on the blockchain and have a gas cost. In addition, our annotations can express some features (such as sums) that could not be directly represented in Solidity. Some examples for pre- and postcondition can be seen above the internal function in lines 6–7 of Figure 3.5.

However, one has to treat preconditions for public functions cautiously. The verifier will simply assume them, but at runtime, the caller can pass in arbitrary arguments, for which the preconditions might not hold. Since they are not compiled and checked runtime, the function will continue to execute, but the assumptions of the verifier do not hold anymore. Note that this is not a limitation for private functions. Since they can only be called from the contract itself, the verifier can check if the preconditions hold at every place where the functions are called.

3.2.1.3 Loop Invariants

As it can be expected, in order to be able to prove properties of contracts that include loops, we allow the loops to be annotated with *invariants*. Loop invariants must hold on entry and must be maintained by the loop. We provide annotations to express invariants over both `for` and `while` loops. These annotations can access the contract state, variables and parameters of their enclosing function, and the loop counter. An example is shown in line 16 of Figure 3.5.

In general, loop invariants can be complex and difficult to write by developers. However, due to the Ethereum execution fees, loops in Solidity contracts tend to be simple and to have a constant bound. For such loops, we expect that developers can specify invariants easily. Furthermore, existing techniques for invariant inference and unrolling could also be applied [FM10; Bla+10], but it is out of scope for this work.

3.2.1.4 Smart Contract-Specific Properties

Working at the level of the Solidity code allows us to extend the specification language with domain-specific properties that are crucial for describing the contract functionality but otherwise not possible to express. For example, a large portion of Ethereum smart contracts manage balances of users with

respect to some assets. It is often natural and desirable to express (as a contract-level invariant) that the amount of the individual assets should be equal to the total supply. One example is the contract invariant of `SimpleBank` in line 1 of Figure 3.2, which succinctly expresses the security of the bank and allows us to identify the reentrancy problem with the contract. The sum function over mappings cannot be expressed at the level of Solidity as the language does not allow iteration over maps. Similarly, the sum is also not expressible in first-order logic. We have, therefore, developed a domain-specific treatment that works for practical examples.

We extended the specification language with a sum function over collections (arrays, mappings). During translation we introduce a shadow variable $\texttt{sum}_\texttt{c}$ that denotes the sum and keeps track of it as the collection `c` is changed. Whenever an item in the collection gets updated by `c[i] = x;`, we also update the shadow variable by $\texttt{sum}_\texttt{c}$ `=` $\texttt{sum}_\texttt{c}$ `- c[i] + x;`. This way, if the developer refers to the sum in an annotation, we simply use the shadow variable in place. This shadow sum is an abstraction of the precise sum that is strong enough to prove many of the properties we are interested in.

3.2.1.5 Correctness

We target functional correctness of contracts with respect to completed[6] transactions and different types of failures. An *expected failure* is a failure due to an exception deliberately thrown to guard from the user (e.g. `require`, `revert`). An *unexpected failure* is any other failure (e.g. `assert`, overflow). We say that a contract is *correct* if transactions that do not fail due to an expected failure (1) also do not fail due to an unexpected failure and (2) satisfy their specification.

Example. *The contract in Figure 3.5 is correct. There are no unexpected failures, and all completing transactions satisfy the specification. Removing, for example, the check in line 10 can result in completed transactions with overflows. As a further example, removing the statement in line 9 will cause the postcondition of `add_to_x` fail. However, removing the call to `add_to_x` in line 15 keeps the contract correct (as it will not change the state and the specification will still hold).*

3.2.2 Translation

In this section, we present the details and properties of our translation from Solidity contracts to the Boogie IVL. The current work and the experiments are based on Solidity version v0.4.25, supporting a majority of its features.[7] We discuss the translation top-down, from contracts and types, through state variables and functions, to statements and expressions.

3.2.2.1 Contracts

The input of the translation is a collection of contracts to be verified, and the output is a single Boogie program with all contracts. We can reason about single and multiple contracts as well. If the code of all contracts is available, we can take all available annotations into account when reasoning. However, this can be unsafe as EVM addresses are not typed (any address can be cast to a contract type) and is to be used with care. We also support inheritance by relying on the compiler to perform flattening and virtual-call disambiguation. The key idea of the translation is to map contract state variables to Boogie global variables and contract functions to Boogie procedures.

[6]Due to the usage of gas, total and partial correctness are equivalent. Furthermore, currently, we do not model gas: running out of gas does not affect correctness as the transaction is reverted. However, we might model it in the future in order to verify liveness properties or to be able to specify an upper bound.

[7]Since then our tool (solc-verify) has been updated to work on v0.5.17 with support for further features.

3.2.2.2 Types

Solidity offers a variety of types, most of them common in programming languages that are easily translated to Boogie types. Booleans are simply mapped to the Boolean type of Boogie. Solidity integers can be either signed and unsigned and can be of different bit-widths ($8, 16, 24, \dots, 256$ bits). In contrast, Boogie has mathematical (unbounded, signed) integers. A simple encoding is to map any Solidity integer to the mathematical integer type of Boogie. This might lead to imprecise analysis, so we also provide a precise encoding by relying on SMT bitvectors, and a pure arithmetic encoding that relies on modular arithmetic (detailed in Section 3.2.2.7). Addresses in Solidity are represented with 160-bit integers, so we also treat them as integers in Boogie. Solidity map types are modeled directly as SMT arrays [McC62; DB09]. Boogie does not have a native array type, so we translate Solidity array types to a pair of an integer length and an SMT array from integers to their element type.[8] Contract reference types are simply represented by addresses. Type checking is already performed by the compiler so only compatible types can be passed around (e.g. as arguments).

There are additional Solidity types that we do not support yet, such as enumerations, tuples, and structures, leaving them for future work.[9] Events (a logging mechanism) are currently ignored as they are not relevant for functional correctness.[10]

3.2.2.3 State Variables

State variables are mapped to global variables in Boogie. However, multiple instances of a contract can be deployed to the blockchain at different addresses. Since aliasing of contract storage is not possible [c11], we model each state variable as a one-dimensional global mapping from contract addresses to their respective type (in essence, treating the blockchain as a heap in a Burstall-Bornat model [Bor00]). For example, the state variable `x` with type `int` at line 2 of Figure 3.6a is translated to the global variable `x` with mapping type `[address]int` at line 1 of Figure 3.6b.

3.2.2.4 Functions

Each function in Solidity is translated to a procedure in Boogie with an additional implicit receiver parameter [Bar+04] called `_this`, which identifies the address of the current contract instance. As an example, consider the `set` function of the Solidity contract `A` in Figure 3.6a. Updating `x` in the Boogie program becomes an update of the map `x` using the receiver parameter `_this`. Consider also the call `a.set(x)` in the Solidity function `setXofA`. The Boogie program first gets the address of the `A` instance corresponding to the current `B` instance using `a[_this]`. Then it passes this address to the receiver parameter of the function `set`.

Functions can be declared view (cannot write state) or pure (cannot read or write state), but these restrictions are checked by the compiler. Additional user-defined function modifiers are a language feature of Solidity to alter or extend the behavior of functions. In practice, modifiers are commonly used to weave in extra checks and instructions to functions. For example, the `pay` function in Figure 3.7a includes the modifier `onlyOwner` (defined in line 4), which performs an extra check before

[8] Note that this is not precise because assigning SMT arrays has deep copy semantics. In contrast, Solidity has different memory locations with different copy semantics (deep or reference). Accurate modeling of memory locations and copy semantics have been added later [c11], but it is out of scope for this thesis.

[9] Support for enumerations, tuples, and structures has since been added [c11], but they are out of scope for this thesis.

[10] Contracts can use events to provide an abstract view of their execution (so that their users do not have to run a full node and replay transactions to get relevant information). In this case events are relevant for functional correctness as the users need to trust their validity. Support for specifying and verifying contracts with events has since been added [a23], but it is out of scope for this thesis.

calling the actual function (denoted by the placeholder _). We simply inline statements of all modifiers of a function to obtain a single Boogie procedure (e.g. pay procedure in Figure 3.7b).

```
1  contract A {
2    int public x;
3    function set(int _x) {
4      x = _x;
5    }
6  }
7  contract B {
8    A a;
9    function setXofA(uint x) {
10     a.set(x);
11   }
12   function getXofA()
13            returns (uint) {
14     return a.x();
15   }
16 }
```

(a) Solidity contract. Each function is public, but the keyword is omitted for brevity.

```
1  var x: [address]int;
2
3  procedure set(_this: address, _x: int) {
4    x := x[_this :=_x];
5  }
6
7  var a: [address]address;
8
9  procedure setXofA(_this: address, x: int) {
10   call set(a[_this], x);
11 }
12
13 procedure getXofA(_this: address)
14                returns (r: int) {
15   r := x[a[_this]];
16 }
```

(b) Boogie representation.

Figure 3.6: Solidity contract and its Boogie translation, illustrating the representation of the blockchain data as a heap and the receiver parameter of functions.

3.2.2.5 Statements and Expressions

Most of the Solidity statements and expressions can be directly mapped to a corresponding statement or expression in Boogie with the same semantics, including variable declarations, conditionals, `while` loops, calls, returns, indexing, unary/binary operations and literals. There are also some statements and expressions that require a simple transformation, such as mapping `for` loops to `while` loops or extracting nested calls and assignments within expressions to separate statements using fresh temporary variables. We currently do not support inline assembly and creating new contracts from within another contract (`new` expressions). Furthermore, the availability of some arithmetic operations depends on the expressiveness of the underlying domain (e.g. bitwise operations, see Section 3.2.2.7).

3.2.2.6 Transactions

Solidity includes Ethereum-specific functions and variables to query and manipulate balances and transactions. Some examples can be seen in Figure 3.7 with the corresponding translation. Each address is associated with its balance, which can be queried using the `balance` member of the address. Correspondingly, we keep track of the balances in a global mapping from addresses to integers (line 1 of Figure 3.7b).

Solidity offers the `msg.sender` field within functions (line 5 of Figure 3.7a) to access the caller address. We map this to Boogie by adding an extra parameter `_msg_sender` of type `address` to each procedure. When a procedure calls another externally, the current receiver address (`_this`) is passed in as the sender. Internal calls are simply translated as jumps in the EVM, which we model by forwarding the original `_msg_sender`.

```
1  contract Wallet {
2    address owner;
3
4    modifier onlyOwner() {
5      require(msg.sender == owner);
6      _;
7    }
8    function receive() payable public {
9      // Actions could be performed here
10   }
11   function pay(address to, uint amount) public onlyOwner {
12     to.transfer(amount);
13   }
14 }
```

(a) Solidity contract.

```
1  var _balance: [address]int;
2  var owner: [address]address;
3
4  procedure receive(_this: address, _msg_sender: address, _msg_value: int) {
5    _balance := _balance[_this := _balance[_this] + _msg_value];
6    // Actions could be performed here
7  }
8  procedure pay(_this: address, _msg_sender: address, _msg_value: int,
9                to: address, amount: int) {
10   assume(_msg_sender == owner[_this]); // Inlined modifier
11   assume(_balance[_this] >= amount);
12   _balance := _balance[_this := _balance[_this] - amount];
13   _balance := _balance[to := _balance[to] + amount];
14 }
```

(b) Boogie representation.

Figure 3.7: A simple wallet, which can receive Ether from anyone but only the owner can make transfers. This example illustrates various Ethereum- and blockchain-specific features in Solidity along with their representation in Boogie.

Solidity functions marked with the `payable` keyword (line 8 of Figure 3.7a) are capable of receiving Ether when called. The amount of Ether received can be queried from the `msg.value` field. We model this in Boogie by including an extra parameter `_msg_value` and updating the global balances map at the beginning of the corresponding Boogie procedure (line 5 of Figure 3.7b). When calling a payable function in Solidity, the amount of Ether to be transferred can be set with the special `value` function (e.g. line 11 of Figure 3.2). We translate this to Boogie by reducing the balance of the caller before making the call and passing the value as the `_msg_value` argument.

The functions `send` and `transfer` are dedicated functions to transfer Ether between addresses. We inline these functions by manipulating the global mapping of balances directly. If the recipient is a contract, a special fallback function is executed, but the gas passed is limited to raising events, which is irrelevant for functional correctness.[11] For example, the transfer in line 12 of Figure 3.7a is

[11] Gas costs of certain write operations were about to change with Constantinople, allowing a reentrancy attack, but it was reverted with the St. Petersburg upgrade: blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement.

mapped to lines 11–13 in Figure 3.7b. The sender not having enough funds is an expected transaction failure, which is modeled with an assumption.

The function `call` can call a function by its name on any address and can also pass arbitrary data. Since there can be an unknown code behind the called address, we treat such cases as an external call that can perform arbitrary computation.[12] We do not support low-level function calls such as `callcode` and `delegatecall` as it is considered dangerous and would require encoding of the EVM details (contract layout, EVM semantics).

Solidity exceptions will undo all changes made to the global state by the current call. Deliberately thrown exceptions (`require`, `revert`, `throw`) are therefore mapped to assumptions in Boogie, which stop the verifier without reporting an error. In contrast, assertions are used to check for internal errors: properly functioning code should never reach a failing assert [Eth18]. Consequently, asserts are mapped to Boogie assertions, causing a reported error when their condition evaluates to false.

3.2.2.7 Encoding of Arithmetic

Integers in Solidity can be signed or unsigned and can be $8, 16, 24, \ldots, 256$ bits wide. Operations over integers in typical contracts are mostly mathematical (addition, subtraction, etc.), but Solidity also supports bitwise operations that are used in real-world contracts to a lesser extent. Depending on the complexity of operations a contract is using, precise reasoning about integers of such large bit-widths can be challenging. Therefore, we support three different encoding modes, which we discuss in the following. An example snippet in Figure 3.8 illustrates the encodings.

```
1 uint8 x = 255;
2 uint8 y = 1;
3 // Assertion holds
4 assert(x + y == 0);
```

(a) Solidity code.

```
1 var x: int;
2 var y: int;
3 x := 255;
4 y := 1;
5 // Assertion does not hold
6 assert(x + y == 0);
```

(b) Boogie encoding with SMT integers.

```
1 var x: bv8;
2 var y: bv8;
3 x := 255bv8;
4 y := 1bv8;
5 // Assertion holds
6 assert(bv8add(x, y) == 0bv8);
```

(c) Boogie encoding with SMT bitvectors.

```
1  var x: int;
2  var y: int;
3  x := 255;
4  y := 1;
5  assume(0 <= x && x <= 255);
6  assume(0 <= y && y <= 255);
7  // Assertion holds
8  assert((if (x + y >= 256)
9          then (x + y - 256)
10         else (x + y)) == 0);
```

(d) Boogie encoding with SMT integers and modular arithmetic.

Figure 3.8: An example Solidity snippet with different arithmetic encodings in Boogie.

Mathematical integers. By default, Boogie treats the integer type as unbounded mathematical integers (Figure 3.8b). This representation allows scalable reasoning with SMT solvers, especially in

[12] Contract invariants are also asserted before external calls as they can perform a callback to the contract.

the case where the constraints are linear [DM06], with much progress being made in recent years on also solving the nonlinear constraints [Rey+17; Jov17]. Therefore, by default, we resort to encoding all Solidity integer types to unbounded integers.

The caveat of this encoding is that it does not support bit-precise operations and that the types and operations are not sound for representing the semantics of Solidity integers (e.g. operations do not overflow). Therefore, verification results should be treated with extreme caution in this case, as they can result in both false alarms and unsound proofs. For example, unsigned integers are guaranteed to be non-negative in Solidity, but the mathematical integers can be possibly negative, causing a false alarm. However, if the contract does not include any bitwise operations, and the programmer is confident that no arithmetic operations go out of range (e.g. by manually checking ranges or using a library like SafeMath [Dou17]), this encoding might be a good fit.

Bitvectors. In order to support exact semantics for Solidity arithmetic, we also provide the encoding that uses the SMT theory of bitvectors to model the integer types and operations over them (Figure 3.8c). This permits to translate almost all Solidity operations (including bitwise operations) to SMT in a fairly straightforward manner but might suffer from scalability issues.

While modern SMT solvers are remarkably efficient on bitvector problems from software applications, the SMT problems arising from Solidity contracts can be much more challenging due to the default integer bit-width of 256 bits. In this setting, if the contract code relies on nonlinear mathematical operations (such as multiplication or division), scalability starts to deteriorate for sizes as small as 16 bits. We provide evidence of this in our evaluation (Section 3.4).

Modular arithmetic. In order to strike a balance between precision and scalability, we also propose an encoding of integers that models Solidity integers as unbounded integers in Boogie but adds additional constraints to model the precise semantics: the allowed value ranges and the wraparound semantics (Figure 3.8d). To track ranges, we associate a type condition (TC) to each integer variable denoting its exact range. Every operation over integer variables is then performed by first assuming the TCs and then performing the corresponding operation in arithmetic modulo the TC range with additional constraints to adjust the results for special cases and signed integers (Figure 3.9). This approach is further sound across all arithmetic expressions since, if the inputs are assumed in the correct range, the results of the operations produce further values in the proper intervals.

The advantage of this approach is that the scalability of reasoning is less dependent on the bit-width, with efficient reasoning also available for example on nonlinear operations over 256-bit integers. We illustrate this in Section 3.4.

Detection of overflows. Neither the EVM nor Solidity perform any checking of the results of arithmetic operations by default. Due to the wraparound semantics of integers, this allows unexpected overflows and underflows to occur undetected (e.g. the infamous BEC token [Dat18]). With an appropriate model of arithmetic, we provide a scalable solution to overflow detection with minimal false alarms.

In general, overflows can be detected by checking the result of every operation after it has been computed. However, reporting every such overflow would result in an overwhelming number of false alarms. For example, it is common practice for Solidity developers to perform arithmetic operations first and then check for overflows manually after the fact (see, e.g. line 10 of Figure 3.5). This practice of overflow detection is an integral part of the SafeMath library [Dou17] that is used in almost all

deployed contracts on the Ethereum blockchain and is part of Solidity best practices [Con18]. Reporting such potential overflows would be a nuisance to the programmer who has already put effort into guarding against it. For example, the potential overflow in line 9 of Figure 3.5 should not be reported because in the very next line the programmer guards the overflow and reverts the transaction.

To reduce the number of false overflow reports, we use the following approach. Whenever an arithmetic computation is performed, we compute the *overflow condition* that captures whether the overflow has occurred (i.e. if the result of the calculation in modular arithmetic is different from the result over unbounded integers). However, instead of immediately checking this condition, it is accumulated in a dedicated Boolean overflow-detection variable. We then check for overflow at the end of every basic block with an assertion. This "delayed checking" gives space to developers to perform manual checking for the overflow (in which case the assertion will not trigger), avoiding false alarms.

$$\begin{aligned}
\mathsf{add}_{U,b}(x,y) &:= \begin{cases} x+y-2^b & \text{if } x+y \geq 2^b \\ x+y & \text{otherwise} \end{cases} \\
\mathsf{add}_{S,b}(x,y) &:= \begin{cases} x+y-2^b & \text{if } x+y > 2^{b-1}-1 \\ x+y+2^b & \text{if } x+y < -2^{b-1} \\ x+y & \text{otherwise} \end{cases} \\
\mathsf{sub}_{U,b}(x,y) &:= \begin{cases} x-y & \text{if } x \geq y \\ x-y+2^b & \text{otherwise} \end{cases} \\
\mathsf{sub}_{S,b}(x,y) &:= \begin{cases} x-y-2^b & \text{if } x-y > 2^{b-1}-1 \\ x-y+2^b & \text{if } x-y < -2^{b-1} \\ x-y & \text{otherwise} \end{cases} \\
\mathsf{mul}_{U,b}(x,y) &:= \begin{cases} x \cdot y \mod 2^b & \text{if } x \cdot y \geq 2^b \\ x \cdot y & \text{otherwise} \end{cases} \\
\mathsf{mul}_{S,b}(x,y) &:= \begin{cases} p-2^b & \text{if } p > 2^{b-1}-1 \\ p & \text{otherwise} \end{cases} \\
&\quad \text{where } p = (\text{if } x \geq 0 \text{ then } x \text{ else } x+2^b) \cdot (\text{if } y \geq 0 \text{ then } y \text{ else } y+2^b) \mod 2^b \\
\mathsf{div}_{U,b}(x,y) &:= x/y \\
\mathsf{div}_{S,b}(x,y) &:= \begin{cases} x/y-2^b & \text{if } x/y > 2^{b-1}-1 \\ x/y+2^b & \text{if } x/y < -2^{b-1} \\ x/y & \text{otherwise} \end{cases} \\
\mathsf{minus}_{U,b}(x) &:= \begin{cases} 0 & \text{if } x = 0 \\ 2^b - x & \text{otherwise} \end{cases} \\
\mathsf{minus}_{S,b}(x) &:= \begin{cases} -2^{b-1} & \text{if } x = -2^{b-1} \\ -x & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3.9: Modeling precise wraparound semantics of arithmetic in modular encoding mode. Each operation is defined for signed (S) and unsigned (U) operands x, y with b bits.

Example. *As an example, consider the function `add_to_x` in Figure 3.5. Its encoding with modular arithmetic and overflow checking can be seen in Figure 3.10. For the simplicity of the presentation, we omit the receiver parameter (`_this`) from state variables, and we are assuming 8 bit integers. A global*

Boolean flag keeps track of the overflow. The function begins by requiring that there are no overflows before calling. It also requires the range assumptions (TC) for the variables it uses and the user-defined precondition (`x == y`). The function ensures that there is no overflow, and its postcondition (`x == y + n`) holds. The postcondition is also checked with modular arithmetic, and an extra ensures clause checks that no overflow occurs in this calculation (`y + n` is the same with integers and modular arithmetic). Increasing `x` by `n` is translated using modular arithmetic, and an extra statement updates the overflow flag if the modular result is different than the integer result. However, this flag is only asserted at the end of the function (`ensures` clause). Thus, the `require` in the original function (translated into an `assume`) can catch the overflow before.

```
1  function modaddS8(x : int, y : int) returns (int) { // Signed, 8 bit add
2    if x + y > 127 then x + y - 256
3    else (if x + y < -128 then x + y + 256
4    else x + y)
5  }
6  var _overflow: bool; // Overflow flag
7  var x: int;
8  var y: int;
9
10 procedure add_to_x(n: int)
11   requires !_overflow;              // No overflow before
12   requires -128 <= x && x <= 127;   // TC for x
13   requires -128 <= y && y <= 127;   // TC for y
14   requires -128 <= n && n <= 127;   // TC for n
15   requires x == y;                  // Precondition
16   ensures !_overflow;               // No overflow after
17   ensures x == modaddS8(y, n);      // Postcondition
18   ensures y + n == modaddS8(y, n); // No overflow in postcondition
19 {
20   _overflow := _overflow || x + n != modaddS8(x, n);
21   x := modaddS8(x, n);
22   assume (x >= y);
23 }
```

Figure 3.10: A (simplified) encoding of the function add_to_x of Figure 3.5, illustrating the delayed overflow check.

3.3 Implementation

We implemented our modular specification and verification approach in an open-source tool named SOLC-VERIFY.[13] An overview of the architecture is shown in Figure 3.11. SOLC-VERIFY extends the original compiler (written in C++) with a translation module for the Boogie IVL, and a Python wrapper script that calls the extended compiler, runs BOOGIE, and maps back the results to the original contracts. The translation and the experiments in this thesis are based on Solidity v0.4.25, but since then, SOLC-VERIFY has been updated to v0.5.17 with support for additional features, most notably precise modeling of memory and reference types [c11].

[13] https://github.com/SRI-CSL/solidity

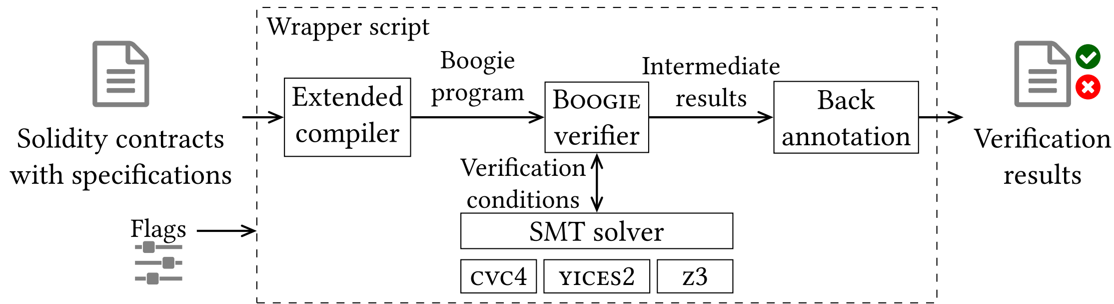


Figure 3.11: Overview of the solc-verify modules. The extended compiler creates a Boogie program from the Solidity contract, which is checked by the Boogie verifier using SMT solvers. Finally, results are mapped back and presented at the Solidity code level.

Translation to Boogie. solc-verify takes a set of Solidity contracts, including specification annotations. It relies on the Solidity compiler that parses the contracts and builds an abstract syntax tree (AST) where names, references, and types are resolved. The compiler assigns a unique identifier (an integer) to each element in the AST. We append this identifier to the name of Boogie declarations (e.g. variables and procedures) to avoid name collisions within the same contract (e.g. state variable and function parameter) and across contracts (e.g. functions with the same name).

The compiler also performs type checking and enforces additional constraints (e.g. visibility). A further advantage of working inside the compiler is that the internal AST contains various extra information, such as the linearized order of constructors or the nature of function calls (internal/external). This makes it easier to follow the semantics of Solidity precisely. We also reuse the compiler to build ASTs for the specification expressions extracted from comments. This way, we can also report meaningful error messages for invalid expressions (e.g. unknown identifiers, wrong types).

solc-verify traverses the internal AST using a visitor and produces a Boogie representation of the program, as discussed in Section 3.2.2. The arithmetic encoding mode (integers, bitvectors or modular; see Section 3.2.2.7) can be specified with a command line flag. The Boogie program is serialized into a temporary file and passed to the Boogie verifier.

Boogie and SMT. Boogie performs modular verification by transforming the program into verification conditions (VCs) and discharging them using SMT solvers (as decribed in Section 3.1.5). By default, Boogie can use z3 [MB08] and cvc4 [Bar+11] but we also extended it[14] to support yices2 [Dut14]. A notable feature of our encoding is that it allows quantifier-free VC generation, permitting to use SMT solvers that do not support quantifiers (e.g. yices2). Boogie reports violated annotations and failing assertions with respect to the Boogie program, and solc-verify maps these errors back to the Solidity code using traceability information. The final output of solc-verify is a list of errors corresponding to the original contracts (e.g. line numbers, function names).

3.4 Evaluation

In this section, we evaluate our modular verification and specification approach on various real-life examples. We formulate the following three research questions for the evaluation.

[14]Our extension has been merged to the main repository: https://github.com/boogie-org/boogie/pull/99.

RQ1 What portion of real-world contracts can SOLC-VERIFY cover with the currently supported language features?
RQ2 Can SOLC-VERIFY find bugs in unannotated contracts by only using assertions and overflows as an implicit specification?
RQ3 Can SOLC-VERIFY find bugs or prove functional correctness of annotated contracts?

Table 3.1 summarizes the evaluation goals, contracts and the observed output. In RQ1 (Section 3.4.1), we observe the coverage of currently-supported language features and scalability by examining the (unannotated) contracts currently deployed on the Ethereum blockchain (available from Etherscan). Then, in RQ2 (Section 3.4.2) we pick a subset of the unannotated contracts and manually check what our approach can report on them. Finally, RQ3 (Section 3.4.3) examines two contracts that had been exploited in the past, and show how our approach could have found the issues, with minimal annotation burden, and prove that the fixed versions of the contracts are correct.

Table 3.1: Summary of the evaluation goals, the models used and the observed output.

	Goal	Contracts	Observed
RQ1	Language coverage	Etherscan	Translation success, runtime
RQ2	Bug finding in unannotated contracts	Etherscan	True and false positives
RQ3	Annotating and checking contracts	Bank, Token	Verification result

We used commit cc913e3 of SOLC-VERIFY[15] and a custom version of BOOGIE[16] that also supports YICES2. Furthermore we used CVC4 commit b396d78, YICES2 version 2.6.1 and Z3 version 4.8.4.

3.4.1 RQ1: Language Coverage

To analyze the coverage of currently-supported language features and the scalability of SOLC-VERIFY, we collected 37531 contracts available on Etherscan.[17] These contracts were compiled with various versions of the Solidity compiler, and not all of them are supported by version 0.4.25 that SOLC-VERIFY used at the time of performing the experiment. We, therefore, selected 7836 contracts that do compile. While these contracts only represent roughly 21% of all the available contracts, we believe that these are the most relevant. Due to the practical immutability of contracts, formal verification should normally be applied during development, which is usually done with the current stable compiler version. Measurements were executed on a machine with 4 pieces of 8 core (3.30 GHz) Intel Xeon E5-4627 v2 CPUs and 1 TB of RAM.

The results of running SOLC-VERIFY on the selected contracts are shown in Table 3.2. Columns correspond to different arithmetic modes, with the last column representing modular arithmetic with overflow checking enabled. The first row shows that roughly 50% of the contracts can be successfully translated to Boogie in each mode. Contracts that cannot be translated contain constructs not yet handled by SOLC-VERIFY, such as structures, enumerations, or special transaction and blockchain members. Some features (e.g. exponentiation, bitwise operations) also depend on the arithmetic mode, resulting in slight differences in feature coverage. The remaining three rows show the number of contracts for which SOLC-VERIFY terminates within 10 seconds with a given SMT solver as a backend. Note that the effectiveness of the different SMT solvers on this set of contracts should be taken with a

[15] https://github.com/SRI-CSL/solidity
[16] https://github.com/dddejan/boogie, commit 928098c
[17] Downloaded from https://etherscan.io, also available at http://csl.sri.com/users/dejan/contracts.tar.gz.

Table 3.2: Etherscan results with different solvers and arithmetic encodings. Each cell represents the number of successfully processed contracts (of 7836 total) and the average execution time per contract.

Arithmetic encoding	Integer	Bitvector	Modular	Mod. overflow
Translated	4096	3919	3926	3926
cvc4	4090 (0.71s)	3837 (0.99s)	3921 (0.72s)	3911 (0.79s)
yices2	3892 (1.15s)	3854 (0.86s)	3903 (0.75s)	3859 (0.87s)
z3	3897 (1.24s)	3831 (1.10s)	3892 (0.87s)	3894 (0.88s)

grain of salt. For example, the bitvector encoding seems to be nearly as efficient as modular arithmetic. However, this is because the assertions in these contracts do not depend on complex (e.g. nonlinear) arithmetic. With more complex invariants, the bitvector encoding becomes infeasible for reasoning, as we demonstrate it with the BEC token example later in Section 3.4.3. The takeaway of these results is that the average execution time per contract is around a second, meaning that solc-verify is applicable and efficient for a significant amount of real-world contracts, but scalability might depend on the complexity of the properties.

3.4.2 RQ2: Unannotated Contracts

The contracts available at Etherscan are *not annotated* and solc-verify can only consider `assert` and `require` statements, and overflows as implicit specification. Furthermore, the ground truth about the contracts (whether they are correct or not) is unknown. Nevertheless, we systematically selected a subset of the contracts and manually checked the results given by solc-verify.

We took all 3897 contracts that solc-verify could translate and process with z3 in integer mode. At first glance we discovered that a majority of the contracts (2754) use the popular SafeMath library [Dou17], which has just recently adopted the proper usage of `assert` and `require`.[18] We updated these contracts to properly guard against user input with `require` (instead of `assert`). Afterwards, we checked for *assertion failures* and *overflows* using solc-verify.

Assertion checking. Surprisingly, only 88 contracts (out of the 3897) contain assertions. solc-verify reported an error for 80 contracts, which we all checked manually. Out of those errors, 78 are clearly false alarms caused by a bad specification – the developer wrote `assert` where `require` should have been used – and fit into one of the following categories:

- Enforcing input validity with assertion (e.g. input arrays are of equal size).
- Enforcing time locks with an assertion (e.g. `now > 100`).
- Enforcing success of functions calls with an assertion (e.g. `addr.call()`).
- Enforcing permissions with an assertion (e.g. checking `msg.sender`).
- Enforcing correct result of arithmetic operations with an assertion.

As described in the Solidity documentation [Eth18] `assert` should only be used to check for internal errors and invariants, and all cases highlighted above should use `require` instead. After replacing the spurious assertions with `require`, solc-verify reports no false alarms.

The two reported errors worth discussing in more detail are illustrated in Figure 3.12. The example in Figure 3.12a is a pre-sale contract that accepts Ether until a sale cap is reached. The invariant of the contract, i.e. that `(raised <= max)` is enforced with a (stronger) assertion at the beginning of

[18] For discussion, see https://github.com/OpenZeppelin/openzeppelin-solidity/issues/1120.

the function. It could be argued that this fits within the mentioned prescribed usage for the `assert` construct. However, as SOLC-VERIFY performs modular analysis, and nothing is assumed about the state before a function call, it will report such an assertion as a potential error. To fix this, the invariant `(raised <= max)` should be specified as a contract invariant, and `require` should be used to check the stronger precondition at function entry (followed by an `assert` at the end of the function).

The example in Figure 3.12b is a token transfer function. The function checks whether the sender has enough balance, and then it transfers the tokens to the recipient. Finally, the assertion checks that no overflow has occurred using an `assert` statement on the result of the addition. As is, SOLC-VERIFY reports an error because increasing the balance of the recipient might overflow. As argued above, if the purpose of the assertion is to guard against overflows, `require` should be used instead. On the other hand, one could argue that for fixed-cap tokens, such an overflow should never occur since no address can hold enough tokens to trigger the overflow. This assumption can be explicitly specified, i.e. by stating a contract invariant `sum(balances) <= cap`. With this invariant, SOLC-VERIFY avoids the false alarm by inferring that overflow is no longer possible.

```
1  uint max    = 1000 ether;
2  uint raised = 0;
3
4  function() payable {
5    assert(raised < max);
6    require(msg.value != 0);
7    require(raised + msg.value
8            <= max);
9    raised += msg.value;
10 }
```

(a) Pre-sale contract example.

```
1  mapping (address => uint) balances;
2
3  function transf(address to, uint val) {
4    require(balances[msg.sender] >= val);
5    require(msg.sender != to);
6    balances[msg.sender] =
7        balances[msg.sender] - val;
8    balances[to] = balances[to] + val;
9    assert(balances[to] >= val);
10 }
```

(b) Token transfer example.

Figure 3.12: Examples of potentially failing assertions reported by SOLC-VERIFY.

Overflow checking. We also checked for overflows and manually verified the results for the 68 contracts (out of 3897) that had at least 100 transactions in the past. SOLC-VERIFY reports 33 errors out of which 29 are false alarms and 4 can be considered as real. All false alarms are due to implicit assumptions on the magnitude of the numbers used. There are 20 false alarms due to missing range assumptions for array lengths causing false overflow alarms for loop counters. For example, in a loop `for (uint i = 0; i < array.length; i++)` SOLC-VERIFY reports that `i++` might overflow. It is reasonable to assume that array lengths remain small due to the gas costs associated with growing an array. After adding these extra assumptions, these errors are no longer reported. Other false alarms are caused by implicit assumptions on Ether balances or time. For example, it is assumed that a counter for the total amount of Ether received by a contract, or multiplying `msg.value` by 20000 cannot overflow because the amount of Ether is limited. Similarly, adding days or even weeks to the current timestamp will not overflow any time soon. We plan to include such implicit assumptions to a limited extent but, in general, it is best if the developer explicitly specifies them. The four issues found that could be considered real are the following:

- A pre-sale contract sets the `hardCap` in its constructor based on a `cap` provided as argument with `hardCap = cap*(10**18)`. Although the constructor is only called once by the deployer, providing a large cap can result in an unintentional overflow.

- A crowd-sale contract sets the unit cost based on the argument `perEther` by calculating `unitCost = 1 ether / (perEther*10**8)`. The problematic function is guarded so that it can only be called by the contract owner. Nevertheless, overflow can happen and can lead to an inconsistent unit price.
- A utility contract for mass distribution of tokens has a function to transfer an array of values to an array of recipients as a batch. The total amount transferred is kept accumulated in a contract counter and can overflow. However, as the counter is not used otherwise, the overflow might be benign.
- A food store contract first calculates the cost based on the bundles ordered, by computing `cost = bundles * price`, where `bundles` is provided by the caller. The function then checks if `msg.value >= cost` holds, but this check can be bypassed with the overflow, opening the door for a potential exploit.

3.4.3 RQ3: Annotated Contracts

While solc-verify can find violations to implicit specifications (assertions, overflows) in unannotated contracts, its main target is to allow developers to check custom, high-level functional properties through annotations. We demonstrate this by annotating two contracts, finding bugs, and proving the correctness of the fixed versions.

Reentrancy detection (DAO). Reentrancy is a common source of vulnerabilities and the cause of the infamous DAO bug [DMH17]. As explained in Section 3.1.3, the `SimpleBank` contract presented in Figure 3.2 suffers from the same reentrancy bug. Using solc-verify, the developer can specify the consistency of the bank contract state through a contract-level invariant, and solc-verify can detect the bug. For example, we can annotate the contract with a property `sum(balances) == this.balance`. As the balance of the contract is deducted before the external call, the contract invariant is violated, and solc-verify reports a (real) error. However, if the user fixes the issue by first reducing the balance of the recipient in the mapping and then transferring the amount, the invariant will hold before making the external call, and solc-verify proves the specification successfully. For both the buggy and correct versions of the contract, the verification with solc-verify is instant.

Overflow detection (BEC token). We now consider the BEC token vulnerability [Dat18] that has also been exploited and resulted in significant financial losses. The relevant part of the contract is shown in Figure 3.13. The contract is a typical token contract, tracking balances of users in terms of their BEC tokens and allowing transfers of tokens between users. The function `batchTransfer` shown in the figure is intended to be used for transferring some value of BEC tokens to a group of recipients in a batch. To do so, the contract multiplies the requested value with the number of recipients. Unfortunately, this multiplication can result in an overflow (line 20), causing the total transfer amount to be invalid (e.g. 0). This allows attackers to "print" large amounts of tokens and send them to other users while keeping their own balance constant. Running solc-verify with the modular encoding of arithmetic successfully detects the overflow issue of BEC token and does not report any other potential overflows. After fixing the contract (line 21), solc-verify shows that no overflows are possible.

We also annotated the BEC contract with a specification that the contract maintains the correct token balances throughout the operation. As before, we add the invariant `totalSupply == sum(balances)` to the contract, and adapt it to the loop invariant. The loop invariant introduces extra complexity as it involves nonlinear arithmetic and illustrates the need for precise reasoning at

large bit-sizes. Running solc-verify on the annotated contract in the bitvector mode does not terminate regardless of the SMT solver used.[19] On the other hand, using modular arithmetic with overflow detection solc-verify discharges all VCs (with 256-bit integers) in seconds for both the buggy and correct version of the contract (with cvc4).

```
library SafeMath {
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a * b;
    require(a == 0 || c / a == b);
    return c;
  }
  // Similar for add, sub, div
}

/// @notice invariant totalSupply == sum(balances)
contract BecToken {
  using SafeMath for uint256;

  uint256 public totalSupply;
  mapping(address => uint256) balances;

  function batchTransfer(address[] _receivers, uint256 _value) public
                            returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value; // Overflow
    // uint256 amount = uint256(cnt).mul(_value); // Correct version
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);
    balances[msg.sender] = balances[msg.sender].sub(amount);
    /// @notice invariant totalSupply == sum(balances) + (cnt - i) * _value
    /// @notice invariant (i <= cnt)
    for (uint i = 0; i < cnt; i++) {
      balances[_receivers[i]] = balances[_receivers[i]].add(_value);
    }
    return true;
  }
}
```

Figure 3.13: Annotated part of the BEC token contract relevant for the “batchOverflow” bug [Dat18]. While the contract uses the SafeMath library for most of its operations, there is a multiplication in line 20 that can overflow.

Other tools. As far as we know, solc-verify is the only available tool that can reason effectively and precisely about Solidity code with specifications. The Solidity compiler includes an experimental SMT checker [AR18], which is currently limited to basic require/assert and overflow checking. For the BEC token, the latest version (v0.5.10) reports every arithmetic operation as a potential overflow, including all false alarms in the SafeMath library. It cannot detect the reentrancy issue in the SimpleBank example because external calls and the revert function is not supported. Furthermore, it incorrectly reports that the condition for revert is always true (possibly because call is skipped and the default return value is false). Zeus [Kal+18] is not available publicly for comparison. VeriSol [Wan+20]

[19] With bit-size of 16 bits, z3 can discharge the VCs in 2295s while other solvers do not terminate.

(as of April 2019) does not support libraries (like `SafeMath`) or the `call` function, which can cause reentrancy so we could not apply it to our examples.

Two notable static analysis tools are Mythril [Mue18] and Slither [FGG19]. Mythril (v0.20.0) correctly reports the overflow issue with the BEC token in 200s, but it also reports all spurious overflows. Mythril detects the reentrancy issue with the bank contract, but it also reports the same issue with the corrected version of the contract. Slither (v0.5.2), on the other hand, has a dedicated DAO-like reentrancy issue check and correctly handles both the buggy and correct version of the bank contract. However, Slither does not support overflow checking and therefore does not detect the BEC token issue.

Our goal, as demonstrated by the annotated examples, is to provide a tool that allows developers to check their own high-level annotations and business logic properties. This makes solc-verify good complementary to other automated verification tools that mainly target well-known vulnerability patterns.

3.5 Related Work

The popularity of blockchain technology and many high-profile attacks and vulnerabilities have put the focus on the need for formal verification for smart contracts [ABC17; HK18; MCJ18; Che+20]. We mention prominent advances relying on vulnerability patterns, theorem provers, finite automata, and SMT, and relate them to our work.

Vulnerability pattern-based approaches. Bhargavan et al. [Bha+16] translate a fragment of Solidity and EVM to F* and use its type and effect system to check for vulnerable patterns and gas boundedness. Grishchenko et al. [GMS18] extend this work on EVM by checking security properties such as call integrity, atomicity, and independence from miner controlled parameters. Securify [Tsa+18] decompiles EVM and infers data- and control-flow dependencies in Datalog to check for compliance and violation patterns. Oyente [Luu+16] is a symbolic execution tool that can check various patterns, including transaction ordering dependency, timestamp dependency, mishandled exceptions, and reentrancy. Maian [Nik+18] uses symbolic analysis with concrete validation over a sequence of invocations to detect fund locking, fund leaking, and contracts that can be killed. Mythril [Mue18] uses symbolic analysis to detect a variety of security vulnerabilities. Slither [FGG19] is a static analysis framework with dedicated vulnerability checkers. Approaches based on vulnerability patterns, like the ones mentioned above, can be effective at discharging specific properties but are limited to built-in patterns (or a domain-specific language [Tsa+18]). Furthermore, as they are mainly EVM-based, it makes reasoning about more general properties difficult. Our approach focuses on Solidity and allows high-level, user-defined properties to be checked effectively.

Theorem prover-based approaches. Kevm [Hil+18] is an executable formal semantics of EVM based on the K framework [RŞ10] including a deductive program verifier to check contracts against given specifications. Hirai [Hir17] formalizes EVM in Lem, a language used by various theorem provers, and proves properties using interactive theorem proving. Scilla [Ser+19] is an intermediate language between smart contracts and bytecode, using the Coq proof assistant for reasoning. Theorem prover-based approaches offer the ability to capture precise, formal semantics of the contracts but can be cumbersome as properties also need to be formalized in the language of the theorem prover. Moreover, user interaction and assistance is usually required impeding usability for contract

developers.[20] In our approach, the developer can specify the properties directly within the contract as Solidity annotations and modular verification is fully automated. Although loop invariants might be required, complex loops are rare in contracts.

Automata-based approaches. FSolidM [ML18] is a finite state machine-based designer for smart contracts that can generate Solidity code. Security features and design patterns (e.g. locking, access control) can be included in the state machine. Abdellatif and Brousmiche [AB18] model contracts and the blockchain manually in BIP and use statistical model checking to simulate uncertainties in the environment. Such model-based approaches are orthogonal to our method, as we are working on the source code directly. This has the advantage that developers do not need to learn a new (modeling) language, and an extra step of transformation (from model to source) is eliminated.

SMT-based approaches. Zeus [Kal+18] translates Solidity to LLVM bitcode and employs existing verifiers such as SeaHorn [Gur+15] and Smack [RE14]. Besides certain vulnerability patterns, it claims to have support for user-defined properties to some extent. However, it is not publicly available for comparison. VeriSol [Wan+20] checks for conformance between workflow policies and smart contract implementations on the Azure blockchain. While the core of their method is a translation to Boogie (similar to ours), it targets a specific problem limited in scope and does not yet support features needed for typical smart contracts (see Section 3.4.3). The Solidity compiler itself also includes a built-in experimental SMT checker [AR18], which executes the body of each function symbolically and checks for implicit specifications, such as assertion failures, dead code and overflows. Their approach is however, limited, by false overflow alarms and missing features (e.g. `call`, `revert`). Furthermore, it has no support for developer-supplied specification beyond `require` and `assert` statements. Some of the challenges they mention in their future work are solved by our approach, including contract-level invariants and the reduced number of false overflow alarms.

3.6 Summary and Future Work

In this thesis, we presented a modular specification and verification approach for smart contracts written in Solidity. We implemented our approach in the solc-verify tool and investigated its applicability on several real-life examples by finding bugs, fixing them, and proving correctness with minimal user effort. My contributions are summarized as follows.

Thesis 3 I defined a modular specification and verification approach for smart contracts by annotating and translating them to an intermediate verification language.

3.1 I adapted existing modular specification constructs to the context of smart contracts.
3.2 I proposed domain-specific annotations for the modular specification and verification of smart contracts.
3.3 I introduced a mapping from the Solidity contract-oriented programming language to the Boogie intermediate verification language.
3.4 I described a modular arithmetic encoding that supports scalable bit-precise reasoning on arithmetic operations.

[20] For an example of the difficulties in manually analyzing even trivial issues, see https://runtimeverification.com/blog/erc-20-verification/.

Joint work. Dejan Jovanović was taking part in this research as my internship supervisor at SRI International. He was also responsible for downloading contracts and running the tool on them. Michael Emmi and Gabriela Ciocarlie also helped with their feedback and advice during our discussions.

Publications. The results and the implementation were presented at the VSTTE 2019 conference [c10]. I also gave a developer-oriented talk about the usage of the tool at the 2020 Solidity Summit.[21] A paper on precise support for reference types (arrays, structs) and different memory locations was published at the ESOP 2020 conference [c11] and was also accepted for presentation at the SMT 2020 workshop.[22] Furthermore, a US patent including (but not limited to) my results was also filed in December 2018 and is currently pending.

Applications. The specification and verification approach is implemented in the open-source solc-verify tool [c10]. solc-verify has been used in a project (TÉT-16-PT) in collaboration with the University of Coimbra. The goal of the project was to inject faults into smart contracts and assess their impact on the system. Results indicated that using solc-verify in the workflow could significantly reduce the number of undetected errors.

Furthermore, solc-verify has also been used to check for behavioral simulation between different smart contracts implementing the same interface [Bei+20].

Future work. While modular verification has a sound mathematical basis, the translation from the high-level Solidity language to Boogie must precisely model the semantics. Hence, an important direction of future work is to formalize the translation and prove its soundness. Some aspects of the memory model and reference types have been formalized [c11], which could be continued for the rest of the translation.

Modular verification relies heavily on loop invariants and function specifications. If arrays are involved, quantifiers are often required, which makes verification conditions undecidable in general. However, quantifiers could be supported in a limited, but decidable fragment called the array property fragment [BMS06]. This would allow the developers to formulate specifications quantified over elements of an array, such as sortedness.

Currently, we only target safety properties, but it would be practical to support liveness queries as well (e.g. to detect vulnerabilities where an attacker can lock funds in a contract). One way we envision this is to introduce special predicates for termination that could be used in the annotations. For example, given a condition over inputs, a transaction has to succeed.

Our internal representation is currently based directly on the Boogie IVL. However, we believe that it could be replaced with a generic, SMT-based representation (e.g. extending the one that we used for formalizing the memory model [c11]). Then this intermediate representation could be translated to alternative verifier backends such as Why3 [FP13] or Dafny [Lei10].

Boogie can report failing annotations, but it is not trivial to reproduce a counterexample, i.e. a concrete trace that leads to the error. We can extract the raw model from the solver, but mapping that back to the Boogie program (and then to the Solidity contract) is not straightforward and would depend on the internal encoding of Boogie. The Corral [LQL12] tool has better support for counterexamples (e.g. recording the values of variables), but it performs bounded model checking. We could first use Boogie to check if there is an error, and if yes, use Corral to find the counterexample.

[21] https://solidity-summit.ethereum.org/

[22] https://fscd-ijcar-2020.org/workshops#SMT

Summary of the Research Results

This dissertation addressed various challenges in applying formal verification in different application domains. We presented extensions to the CEGAR-based reachability analysis of Petri Nets to lift its expressive power and to increase the number of conclusive answers. We developed various novel strategies for CEGAR-based software model checking to make it more efficient in terms of execution time. We proposed a modular specification and verification approach for smart contracts that is expressive and efficient. This final chapter summarizes our work and highlights the contributions and the key ideas.

Thesis 1: Extensions to the CEGAR Approach on Petri Nets

Petri nets are widely used for modeling concurrent and asynchronous systems. Many interesting properties of the system can be reduced to reachability analysis, i.e. checking if a given marking is reachable from the initial state of the net. While reachability is decidable, it is a complex problem that can be addressed in different ways. The basis of our work is an algorithm that uses the state equation of Petri nets as an over-approximation to reachability. The state equation is an efficient technique (only depending on the structure of the net) but is only a necessary condition. Thus, in the case of infeasible solutions, the algorithm makes the approximation more precise by extending the equation with additional constraints in a CEGAR fashion. This thesis focused on the expressive power and the conclusive answers of the algorithm.

We extended the algorithm to be able to handle *reachability of predicates*, an important generalization of reachability, where the target marking can be specified by a set of linear inequalities. This was done by transforming the linear inequalities over the marking to constraints over transitions that can be directly handled by the algorithm. We also proposed an extension to support Petri nets with *inhibitor arcs* that can test the emptiness of places. Since these arcs do not appear in the state equation, we had to adapt the refinement step of the algorithm to be able to add extra constraints to the equation corresponding to inhibitor arcs. Inhibitor arcs lift the expressive power of Petri nets to be Turing complete, making reachability undecidable. Nevertheless, we presented examples where the algorithm works.

From the side of conclusive answers, our prior work showed a whole subclass of nets where the algorithm could not solve reachability. The reason for this was that the algorithm only tries to involve invariants directly connected to a place that needs tokens. The key idea of our approach was to involve *indirectly connected* (“distant”) *invariants* transitively with a new iteration strategy. We also experimented with breadth- and depth-first search strategies and proposed their combination as a

hybrid strategy. The basis of the *hybrid search strategy* is a partial order between the solutions based on the transitions that could not fire: both minimal and maximal solutions are of interest. While the algorithm can still give inconclusive answers with the extensions, we discussed its theoretical limitations. My contributions in this thesis are summarized as follows.

> **Thesis 1** I proposed extensions and improvements to the CEGAR-based reachability analysis of Petri nets, lifting its expressive power and increasing the amount of conclusive answers.
> 1.1 I generalized the algorithm to be able to solve reachability of predicates, where the target state to be reached can be described with a set of linear constraints.
> 1.2 I extended the algorithm to be able to handle Petri nets with inhibitor arcs, raising its expressive power.
> 1.3 I defined the concept of distant invariants and proposed a new iteration strategy, which extended the kind of problems the algorithm could solve.
> 1.4 I defined a new ordering between partial solutions and a corresponding hybrid search strategy that can speed up the convergence of the algorithm without losing solutions.

Publications related to this thesis are [j1], [j2], [c4], [c5] and [c7].

Thesis 2: Efficient Strategies for CEGAR-based Software Model Checking

Embedded software code written in lower level languages is often modeled using control-flow automata. Software model checking can prove interesting properties of programs by reducing the problem to checking the reachability of a distinguished error location. However, programs often have many variables with rich domains, yielding a state space that cannot be managed explicitly. The basis of our work is a generic, abstraction-based framework that incorporates explicit-value analysis and predicate abstraction. The former only tracks a subset of the program variables while the latter keeps track of facts and relationships using logical formulas. The set of tracked variables and predicates is extended via abstraction refinement in a CEGAR fashion. This thesis focused on the efficiency of CEGAR-based software model checking by proposing novel extensions and combinations of existing strategies.

By default, explicit-value abstraction always calculates a single successor state: if an expression cannot be deterministically evaluated, it is treated as unknown. We extended this domain by trying to *explicitly enumerate* a predefined, configurable number of successor states in such cases. While this has a minimal performance penalty, it can be compensated later by the increased precision. We also adapted a search strategy that uses structural information about the program to guide the search in the abstract state space towards counterexamples. The key idea was to use the syntactical *distance to the error location* in the program as an under-approximating metric when processing states from the queue. This is also beneficial for correct models, as intermediate steps of CEGAR do have counterexamples.

We improved abstraction refinement with a novel interpolation strategy based on processing the spurious counterexample *backwards from the erroneous state*. This strategy can trace back the reason of infeasibility to the earliest point in the abstract state space, yielding a faster convergence to the appropriate precision. Furthermore, we proposed an approach that does not stop building abstraction at the first counterexample, but rather collects all of them. Then, during refinement, *all counterexamples are checked*, and a minimal subset is calculated for refinement. This can also yield a faster convergence and more effective refinements. My contributions in this thesis are summarized as follows.

Thesis 2 I proposed various improvements and strategies to CEGAR-based software model checking, increasing the efficiency of the algorithm.

2.1 I generalized explicit-value analysis to be able to enumerate a predefined, configurable number of successor states, improving its precision, but avoiding state space explosion.

2.2 I adapted a search strategy to the context of CEGAR that estimates the distance from the erroneous state in the abstract state space based on the structure of the software, efficiently guiding exploration towards counterexamples.

2.3 I introduced an interpolation strategy based on backward reachability, that traces back the reason of infeasibility to the earliest point in the program, yielding a faster refinement convergence.

2.4 I described an approach for refinement based on multiple counterexamples, which allows exchanging information between counterexamples and provides better quality refinements.

Publications related to this thesis are [j3], [c6], [c8], [c9], [e12] and [e13].

Thesis 3: Modular Specification and Verification of Smart Contracts

Solidity is a widely used language to write smart contracts to be deployed on the Ethereum blockchain. While there has been a great attention in using static analysis and theorem proving to verify contracts, not much work has been done on automated verification of high-level, functional properties. Due to the transactional behavior of the blockchain, modular specification and verification is a natural way of describing and proving functional properties of smart contracts. Boogie is an intermediate verification language (IVL) with features for modular specification and various verifier backends. This thesis focused on proposing a modular verification and specification approach for smart contracts that is expressive and efficient.

We adapted various existing *specification constructs* to smart contracts, including assertions, pre- and postconditions, and invariants. These specifications are in-code annotations written in a subset of Solidity. Furthermore, we also introduced *domain specific properties* (e.g. sum of collections) that are not directly expressible in Solidity or in the verification logic. Such properties are crucial for many applications in the blockchain domain (e.g. tokens, wallets).

We developed a *translation* from annotated Solidity contracts to the Boogie IVL in order to support modular verification. The key idea of our approach was to model state variables as one dimensional heaps and contract functions as Boogie procedures. While many elements of the translation were straightforward, there were challenges related to blockchain specific constructs (e.g. transactional behavior). One particularly interesting aspect of smart contracts is that they often involve computation over large bit-widths (up to 256 bits). We introduced a *modular encoding of arithmetic* based on SMT integers that models the precise wraparound semantics with range assumptions and modulo operations. This allowed scalable reasoning on variables up to large bit-widths even in the presence of nonlinear arithmetic. My contributions in this thesis are summarized as follows.

Thesis 3 I defined a modular specification and verification approach for smart contracts by annotating and translating them to an intermediate verification language.

3.1 I adapted existing modular specification constructs to the context of smart contracts.

3.2 I proposed domain-specific annotations for the modular specification and verification of smart contracts.

> 3.3 I introduced a mapping from the Solidity contract-oriented programming language to the Boogie intermediate verification language.
> 3.4 I described a modular arithmetic encoding that supports scalable bit-precise reasoning on arithmetic operations.

Publications related to this thesis are [c10] and [c11].

Publications

Number of publications:	19
Number of peer-reviewed journal papers (written in English):	3
Number of articles in journals indexed by WoS or Scopus:	3
Number of publications (in English) with at least 50% contribution of the author:	8
Number of peer-reviewed publications:	18
Number of independent citations:	30

Publications Linked to the Theses

	Journal papers	International conference and workshop papers	Local events
Thesis 1	[j1] [j2]	[c4] [c5] [c7]	—
Thesis 2	[j3]	[c6] [c8] [c9]	[e12] [e13]
Thesis 3	—	[c10] [c11]	—

This classification follows the faculty's Ph.D. publication score system.

Journal Papers

[j1] Ákos Hajdu, András Vörös, Tamás Bartha, and Zoltán Mártonka. Extensions to the CEGAR approach on Petri nets. *Acta Cybernetica* 21(3), 2014, pp. 401–417. DOI: 10.14232/actacyb.21.3.2014.8.

[j2] András Vörös, Dániel Darvas, Ákos Hajdu, Attila Klenik, Kristóf Marussy, Vince Molnár, Tamás Bartha, and István Majzik. Industrial applications of the PetriDotNet modelling and analysis tool. *Science of Computer Programming* 157, 2018, pp. 17–40. DOI: 10.1016/j.scico.2017.09.003.

[j3] Ákos Hajdu and Zoltán Micskei. Efficient strategies for CEGAR-based model checking. *Journal of Automated Reasoning* Online first, 2019. DOI: 10.1007/s10817-019-09535-x.

International Conference and Workshop Papers

[c4] Ákos Hajdu, András Vörös, Tamás Bartha, and Zoltán Mártonka. Extensions to the CEGAR approach on Petri nets. In: *Proceedings of the 13th Symposium on Programming Languages and Software Tools*, pp. 274–288. University of Szeged, 2013.

[c5] Ákos Hajdu, András Vörös, and Tamás Bartha. New search strategies for the Petri net CEGAR approach. In: *Application and Theory of Petri Nets and Concurrency*, Lecture Notes in Computer Science, vol. 9115, pp. 309–328. Springer, 2015. doi: 10.1007/978-3-319-19488-2_16.

[c6] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. In: *Formal Techniques for Distributed Objects, Components and Systems*, Lecture Notes in Computer Science, vol. 9688, pp. 158–174. Springer, 2016. doi: 10.1007/978-3-319-39570-8_11.

[c7] András Vörös, Dániel Darvas, Vince Molnár, Attila Klenik, Ákos Hajdu, Attila Jámbor, Tamás Bartha, and István Majzik. PetriDotNet 1.5: extensible Petri net editor and analyser for education and research. In: *Application and Theory of Petri Nets and Concurrency*, Lecture Notes in Computer Science, vol. 9698, pp. 123–132. Springer, 2016. doi: 10.1007/978-3-319-39086-4_9.

[c8] Gyula Sallai, Ákos Hajdu, Tamás Tóth, and Zoltán Micskei. Towards evaluating size reduction techniques for software model checking. In: *Proceedings of the Fifth International Workshop on Verification and Program Transformation*, Electronic Proceedings in Theoretical Computer Science, vol. 253, pp. 75–91. Open Publishing Association, 2017. doi: 10.4204/EPTCS.253.7.

[c9] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In: *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pp. 176–179. 2017. doi: 10.23919/FMCAD.2017.8102257.

[c10] Ákos Hajdu and Dejan Jovanović. Solc-verify: a modular verifier for Solidity smart contracts. In: *Verified Software. Theories, Tools, and Experiments*, Lecture Notes in Computer Science, vol. 12301, pp. 161–179. Springer, 2020. doi: 10.1007/978-3-030-41600-3_11.

[c11] Ákos Hajdu and Dejan Jovanović. SMT-friendly formalization of the Solidity memory model. In: *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 12075, pp. 224–250. Springer, 2020. doi: 10.1007/978-3-030-44914-8_9.

Local Event Papers

[e12] Ákos Hajdu and Zoltán Micskei. Exploratory analysis of the performance of a configurable CEGAR framework. In: *Proceedings of the 24th PhD Mini-Symposium*, pp. 34–37. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2017. doi: 10.5281/zenodo.291895.

[e13] Ákos Hajdu and Zoltán Micskei. A preliminary analysis on the effect of randomness in a CEGAR framework. In: *Proceedings of the 25th PhD Mini-Symposium*, pp. 32–35. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2018. doi: 10.5281/zenodo.1219261.

Additional Publications (Not Linked to Theses)

International Conference and Workshop Papers

[c14] Ákos Hajdu, Róbert Németh, Szilvia Varró-Gyapay, and András Vörös. Petri net based trajectory optimization. In: *ASCONIKK 2014: Extended Abstracts. Future Internet Services*, pp. 11–19. University of Pannonia, 2014.

[c15] Bence Czipó, Ákos Hajdu, Tamás Tóth, and István Majzik. Exploiting hierarchy in the abstraction-based verification of statecharts using SMT solvers. In: *Proceedings of the 14th International Workshop on Formal Engineering Approaches to Software Components and Architectures*, Electronic Proceedings in Theoretical Computer Science, vol. 245, pp. 31–45. Open Publishing Association, 2017. DOI: 10.4204/EPTCS.245.3.

[c16] Rebeka Farkas, Tamás Tóth, Ákos Hajdu, and András Vörös. Backward reachability analysis for timed automata with data variables. In: *Proceedings of the 18th International Workshop on Automated Verification of Critical Systems*, Electronic Communications of the EASST, vol. 76, pp. 1–20. EASST, 2018. DOI: 10.14279/tuj.eceasst.76.1076.

Local Event Papers

[e17] Rebeka Farkas and Ákos Hajdu. Activity-based abstraction refinement for timed systems. In: *Proceedings of the 24th PhD Mini-Symposium*, pp. 18–21. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2017. DOI: 10.5281/zenodo.291891.

[e18] Viktória Dorina Bajkai and Ákos Hajdu. Software model checking with a combination of explicit values and predicates. In: *Proceedings of the 26th PhD Mini-Symposium*, pp. 4–7. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2019. DOI: 10.5281/zenodo.2597969.

Technical Reports

[r19] Ákos Hajdu. *Making the TTreeReader interface more accessible*. Tech. rep. CERN-STUDENTS-Note-2015-039. European Organization for Nuclear Research (CERN), Aug. 2015.

Additional Work

[a20] Ákos Hajdu. Extensions to the CEGAR Approach on Petri Nets. Bachelor's thesis. Budapest University of Technology and Economics, 2013.

[a21] Ákos Hajdu. A Survey on CEGAR-based Model Checking. Master's thesis. Budapest University of Technology and Economics, 2015.

[a22] Ákos Hajdu and Zoltán Micskei. *Supplementary Material for the paper "Efficient Strategies for CEGAR-based Model Checking"*. 2018. DOI: 10.5281/zenodo.1252784. (Dataset).

[a23] Ákos Hajdu, Dejan Jovanović, and Gabriela Ciocarlie. Formal Specification and Verification of Solidity Contracts with Events. 2020. URL: https://arxiv.org/abs/2005.10382. (Preprint).

Bibliography

[AB18] Tesnim Abdellatif and Kei-Leo Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. In: *Proceedings of the 9th IFIP International Conference on New Technologies, Mobility and Security*, pp. 1–5. IEEE, 2018. DOI: 10.1109/NTMS.2018.8328737.

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. In: *Principles of Security and Trust*, Lecture Notes in Computer Science, vol. 10204, pp. 164–186. Springer, 2017. DOI: 10.1007/978-3-662-54455-6_8.

[AGC12] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig interpretation. In: *Static Analysis*, Lecture Notes in Computer Science, vol. 7460, pp. 300–316. Springer, 2012. DOI: 10.1007/978-3-642-33125-1_21.

[Alb+12] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. UFO: a framework for abstraction- and interpolation-based software verification. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 7358, pp. 672–678. Springer, 2012. DOI: 10.1007/978-3-642-31424-7_48.

[Alb+14] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design* 45(1), 2014, pp. 63–109. DOI: 10.1007/s10703-014-0209-9.

[Alb15] Aws Albarghouthi. Software Verification with Program-Graph Interpolation and Abstraction. PhD thesis. University of Toronto, 2015.

[Alu99] Rajeev Alur. Timed automata. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 1633, pp. 8–22. Springer, 1999. DOI: 10.1007/3-540-48683-6_3.

[Amp+16] Elvio Gilberto Amparore, Gianfranco Balbo, Marco Beccuti, Susanna Donatelli, and Giuliana Franceschinis. 30 years of GreatSPN. In: *Principles of Performance and Reliability Modeling and Evaluation*, Springer Series in Reliability Engineering, pp. 227–254. Springer, 2016. DOI: 10.1007/978-3-319-30599-8_9.

[Amp+19] Elvio Amparore et al. Presentation of the 9th edition of the Model Checking Contest. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 11429, pp. 50–68. Springer, 2019. DOI: 10.1007/978-3-030-17502-3_4.

[Ape+13] Sven Apel, Dirk Beyer, Karlheinz Friedberger, Franco Raimondi, and Alexander von Rhein. Domain types: abstract-domain selection based on variable usage. In: *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, vol. 8244, pp. 262–278. Springer, 2013. DOI: 10.1007/978-3-319-03077-7_18.

[AR18] Leonardo Alt and Christian Reitwiessner. SMT-based verification of Solidity smart contracts. In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Lecture Notes in Computer Science, vol. 11247, pp. 376–388. Springer, 2018. DOI: 10.1007/978-3-030-03427-6_28.

[AW18] Andreas Antonopoulos and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly Media, 2018.

[Baj18] Viktória Dorina Bajkai. Combining Abstract Domains for Software Model Checking. Bachelor's Thesis. Budapest University of Technology and Economics, 2018.

[Bal04] Thomas Ball. *Formalizing Counterexample-driven Refinement with Weakest Preconditions*. Tech. rep. MSR-TR-2004-134. Microsoft Research, 2004.

[Bar+04] Michael Barnett, Robert DeLine, Manuel Fähndrich, K Rustan M Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology* 3(6), 2004, pp. 27–56.

[Bar+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: a modular reusable verifier for object-oriented programs. In: *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer, 2006. DOI: 10.1007/11804192_17.

[Bar+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_14.

[BDW15] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 9206, pp. 622–640. Springer, 2015. DOI: 10.1007/978-3-319-21690-4_42.

[BDW18] Dirk Beyer, Matthias Dangl, and Philipp Wendler. A unifying view on SMT-based software verification. *Journal of Automated Reasoning* 60(3), 2018, pp. 299–335. DOI: 10.1007/s10817-017-9432-6.

[Bei+20] Sidi Mohamed Beillahi, Gabriela Ciocarlie, Michael Emmi, and Constantin Enea. Behavioral simulation for smart contracts. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 470–486. ACM, 2020. DOI: 10.1145/3385412.3386022.

[Bey+07] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer* 9(5), 2007, pp. 505–525. DOI: 10.1007/s10009-007-0044-z.

[Bey+09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In: *Proceedings of the 2009 Conference on Formal Methods in Computer-Aided Design*, pp. 25–32. IEEE, 2009. DOI: 10.1109/FMCAD.2009.5351147.

[Bey12] Dirk Beyer. Competition on software verification. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 7214, pp. 504–524. Springer, 2012. DOI: 10.1007/978-3-642-28756-5_38.

[Bey15] Dirk Beyer. Software verification and verifiable witnesses. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 9035, pp. 401–416. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_31.

[Bey16] Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 9636, pp. 887–904. Springer, 2016. DOI: 10.1007/978-3-662-49674-9_55.

[Bey17] Dirk Beyer. Software verification with validation of results. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 10206, pp. 331–349. Springer, 2017. DOI: 10.1007/978-3-662-54580-5_20.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.smt-lib.org. 2016.

[Bha+16] Karthikeyan Bhargavan et al. Formal verification of smart contracts: short paper. In: *ACM Workshop on Programming Languages and Analysis for Security*, pp. 91–96. ACM, 2016. DOI: 10.1145/2993600.2993611.

[BHM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. IOS press, 2009.

[BHT07] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: concretizing the convergence of model checking and program analysis. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 4590, pp. 504–518. Springer, 2007. DOI: 10.1007/978-3-540-73368-3_51.

[BHT08] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 29–38. IEEE, 2008. DOI: 10.1109/ASE.2008.13.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer, 1999. DOI: 10.1007/3-540-49059-0_14.

[Bie07] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Tech. rep. Report 07/1. Institute for Formal Models and Verification, Johannes Kepler University, 2007.

[BK11] Dirk Beyer and M Erkan Keremoglu. CPAchecker: a tool for configurable software verification. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_16.

[BKW10] Dirk Beyer, M Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pp. 189–198. FMCAD Inc., 2010.

[BL05] Mike Barnett and K Rustan M Leino. Weakest-precondition of unstructured programs. In: *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 82–87. ACM, 2005. DOI: 10.1145/1108792.1108813.

[BL13] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In: *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, vol. 7793, pp. 146–162. Springer, 2013. DOI: 10.1007/978-3-642-37057-1_11.

[Bla+10] Régis Blanc, Thomas A Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: algebraic bound computation for loops. In: *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, vol. 6355, pp. 103–118. Springer, 2010. DOI: 10.1007/978-3-642-17511-4_7.

[BLS05] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The Spec# programming system: an overview. In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Lecture Notes in Computer Science, vol. 3362, pp. 49–69. Springer, 2005. DOI: 10.1007/978-3-540-30569-9_3.

[BLW15a] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement selection. In: *Model Checking Software*, Lecture Notes in Computer Science, vol. 9232, pp. 20–38. Springer, 2015. DOI: 10.1007/978-3-319-23404-5_3.

[BLW15b] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Sliced path prefixes: an effective method to enable refinement selection. In: *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, vol. 9039, pp. 228–243. Springer, 2015. DOI: 10.1007/978-3-319-19195-9_15.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21(1), 2019, pp. 1–29. DOI: 10.1007/s10009-017-0469-y.

[BM07] Aaron R Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification.* Springer, 2007.

[BMS06] Aaron R Bradley, Zohar Manna, and Henny B Sipma. What's decidable about arrays? In: *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, vol. 3855, pp. 427–442. Springer, 2006. DOI: 10.1007/11609773_28.

[Bøn+18] Frederik Bønneland, Jakob Dyhr, Peter G Jensen, Mads Johannsen, and Jiří Srba. Simplification of CTL formulae for efficient model checking of Petri nets. In: *Application and Theory of Petri Nets and Concurrency*, Lecture Notes in Computer Science, vol. 10877, pp. 143–163. Springer, 2018. DOI: 10.1007/978-3-319-91268-4_8.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic. In: *Mathematics of Program Construction*, Lecture Notes in Computer Science, vol. 1837, pp. 102–126. 2000. DOI: 10.1007/10722010_8.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In: *Formal Methods in Programming and Their Applications*, Lecture Notes in Computer Science, vol. 735, pp. 128–141. Springer, 1993. DOI: 10.1007/BFb0039704.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram Rajamani. Boolean and Cartesian abstraction for model checking C programs. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 2031, pp. 268–283. Springer, 2001. DOI: 10.1007/3-540-45319-9_19.

[BR01] Thomas Ball and Sriram Rajamani. The Slam toolkit. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 2102, pp. 260–264. Springer, 2001. DOI: 10.1007/3-540-44585-4_25.

[BR02] Thomas Ball and Sriram Rajamani. *Generating Abstract Explanations of Spurious Counterexamples in C Programs*. Tech. rep. MSR-TR-2002-09. Microsoft Research, 2002.

[Bra11] Aaron R Bradley. SAT-based model checking without unrolling. In: *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer, 2011. DOI: 10.1007/978-3-642-18275-4_7.

[Brü+07] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. Slicing abstractions. In: *International Symposium on Fundamentals of Software Engineering*, Lecture Notes in Computer Science, vol. 4767, pp. 17–32. Springer, 2007. DOI: 10.1007/978-3-540-75698-9_2.

[Bry86] Randal E Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* 100(8), 1986, pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[BS08] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 443–446. IEEE, 2008. DOI: 10.1109/ASE.2008.69.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In: *Handbook of Model Checking*, pp. 305–343. Springer, 2018. DOI: 10.1007/978-3-319-10575-8_11.

[Bur+90] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn 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. 1990. DOI: 10.1109/LICS.1990.113767.

[Bus02] Nadia Busi. Analysis issues in Petri nets with inhibitor arcs. *Theoretical Computer Science* 275(1-2), 2002, pp. 127–177. DOI: 10.1016/S0304-3975(01)00127-X.

[BW12] Dirk Beyer and Philipp Wendler. Algorithms for software model checking: Predicate abstraction vs. Impact. In: *Proceedings of the 2012 Conference on Formal Methods in Computer-Aided Design*, pp. 106–113. IEEE, 2012.

[Cab+16] Gianpiero Cabodi, Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Danilo Vendraminetto, Armin Biere, Keijo Heljanko, and Jason Baumgartner. Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks. *Journal on Satisfiability, Boolean Modeling and Computation* 9, 2016, pp. 135–172.

[Cal+15] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In: *NASA Formal Methods*, Lecture Notes in Computer Science, vol. 9058, pp. 3–11. Springer, 2015. DOI: 10.1007/978-3-319-17524-9_1.

[Cav+07] Roberto Cavada, Alessandro Cimatti, Anders Franzén, Krishnamani Kalyanasundaram, Marco Roveri, and R K Shyamasundar. Computing predicate abstractions by integrating BDDs and SMT solvers. In: *Proceedings of the 2007 Conference on Formal Methods in Computer-Aided Design*, pp. 69–76. IEEE, 2007. DOI: 10.1109/FMCAD.2007.18.

[CC77] Patrick Cousot and Radhia 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. 1977. DOI: 10.1145/512950.512973.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pp. 209–224. USENIX Association, 2008.

[CE82] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: *Logics of Programs*, Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer, 1982. DOI: 10.1007/BFb0025774.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems* 16(5), 1994, pp. 1512–1542. DOI: 10.1145/186025.186051.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 1999.

[CGS04] Edmund M Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23(7), 2004, pp. 1113–1123. DOI: 10.1109/TCAD.2004.829807.

[Che+20] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. Defining smart contract defects on Ethereum. *IEEE Transactions on Software Engineering*, 2020. (Early access).

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: an interpolating SMT solver. In: *Model Checking Software*, Lecture Notes in Computer Science, vol. 7385, pp. 248–254. Springer, 2012. DOI: 10.1007/978-3-642-31759-0_19.

[Cho+20] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R Tuttle. Code level model-checking in the software development workflow. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020. (In press).

[Chr99] Piotr Chrząstowski-Wachtel. Testing undecidability of the reachability in Petri nets with the help of 10th Hilbert problem. In: *Application and Theory of Petri Nets 1999*, Lecture Notes in Computer Science, vol. 1639, pp. 268–281. Springer, 1999. DOI: 10.1007/3-540-48745-X_16.

[Chu36] Alonzo Church. A note on the Entscheidungsproblem. *The Journal of Symbolic Logic* 1(1), 1936, pp. 40–41.

[Cim+13] Alessandro Cimatti, Alberto Griggio, BastiaanJoost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer, 2013. DOI: 10.1007/978-3-642-36742-7_7.

[Cla+03] Edmund M Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM* 50(5), 2003, pp. 752–794. DOI: 10.1145/876638.876643.

[Cla+05] Edmund M Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SatAbs: SAT-based predicate abstraction for ANSI-C. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 3440, pp. 570–574. Springer, 2005. DOI: 10.1007/978-3-540-31980-1_40.

[Cla+18] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick P Bloem. *Handbook of model checking*. Springer, 2018. DOI: 10.1007/978-3-319-10575-8.

[CLS01] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: an efficient iteration strategy for symbolic state-space generation. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 2031, pp. 328–342. Springer, 2001. DOI: 10.1007/3-540-45319-9_23.

[CNQ11] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Interpolation sequences revisited. In: *2011 Design, Automation and Test in Europe*, pp. 1–6. IEEE, 2011. DOI: 10.1109/DATE.2011.5763056.

[Con18] ConsenSys. *Ethereum Smart Contract Security Best Practices*. 2018. URL: https://consensys.github.io/smart-contract-best-practices/.

[Coo71] Stephen 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.

[Cor17] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. 2017. URL: https://www.R-project.org/.

[Cra57] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic* 22(03), 1957, pp. 269–285.

[CT05] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In: *Formal Techniques for Networked and Distributed Systems*, Lecture Notes in Computer Science, vol. 3731, pp. 443–457. Springer, 2005. DOI: 10.1007/11562436_32.

[CT93] Gianfranco Ciardo and Kishor S Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation* 18(1), 1993, pp. 37–59. DOI: 10.1016/0166-5316(93)90026-Q.

[CU05] Sinan Cayir and Mürvet Ucer. An algorithm to compute a basis of Petri net invariants. In: *Proceedings of the 4th ELECO International Conference on Electrical and Electronics Engineering*, UCTEA, 2005.

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13(4), 1991, pp. 451–490. DOI: 10.1145/115372.115320.

[Cze+17] Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Predicting rankings of software verification tools. In: *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*, pp. 23–26. ACM, 2017. DOI: 10.1145/3121257.3121262.

[Cze+19] Wojciech Czerwiundefinedski, Sławomir Lasota, Ranko Laziundefined, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for Petri nets is not elementary. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pp. 24–33. ACM, 2019. DOI: 10.1145/3313276.3316369.

[Czi16] Bence Czipó. Hierarchical Abstraction for the Verification of State-based Systems. Bachelor's Thesis. Budapest University of Technology and Economics, 2016.

[CZJ12] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. Ten years of saturation: a Petri net perspective. In: *Transactions on Petri Nets and Other Models of Concurrency*, Lecture Notes in Computer Science, vol. 2031, pp. 51–95. Springer, 2012. doi: 10.1007/978-3-642-29072-5_3.

[Dar+18] Priyanka Darke, Sumanth Prabhu, Bharti Chimdyalwar, Avriti Chauhan, Shrawan Kumar, Animesh Basakchowdhury, R Venkatesh, Advaita Datar, and Raveendra Kumar Medicherla. VeriAbs: verification by abstraction and test generation. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 10806, pp. 457–462. Springer, 2018. doi: 10.1007/978-3-319-89963-3_32.

[Dar14] Dániel Darvas. Incremental extension of the saturation algorithm-based bounded model checking of Petri nets. Master's Thesis. Budapest University of Technology and Economics, 2014.

[Dat18] NIST National Vulnerability Database. *CVE-2018-10299: Beauty Ecosystem Coin (BEC) "batchOverflow" issue*. 2018. url: https://nvd.nist.gov/vuln/detail/CVE-2018-10299.

[DB09] Leonardo De Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In: *Proceedings of the 2009 Conference on Formal Methods in Computer-Aided Design*, pp. 45–52. 2009. doi: 10.1109/FMCAD.2009.5351142.

[DBM19] Dániel Darvas, Enrique Blanco Viñuela, and Vince Molnár. PLCverif re-engineered: An open platform for the formal analysis of PLC programs. In: *Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems*, JACoW, 2019.

[Dem+17] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. Empirical software metrics for benchmarking of verification tools. *Formal Methods in System Design* 50(2), 2017, pp. 289–316. doi: 10.1007/s10703-016-0264-5.

[DFB15] Dániel Darvas, Borja Fernández Adiego, and Enrique Blanco Viñuela. PLCverif: A tool to verify PLC programs based on model checking techniques. In: *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, pp. 911–914. JACoW, 2015. doi: 10.18429/JACoW-ICALEPCS2015-WEPGF092.

[Die+17] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. Craig vs. Newton in software model checking. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 487–497. ACM, 2017. doi: 10.1145/3106237.3106307.

[Dij71] Edsger Wybe Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica* 1(2), 1971, pp. 115–138. doi: 10.1007/BF00289519.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27(7), 2008, pp. 1165–1178. doi: 10.1109/TCAD.2008.923410.

[DL05] Robert DeLine and K Rustan M Leino. *BoogiePL: A typed procedural language for checking object-oriented programs*. Tech. rep. MSR-TR-2005-70. Microsoft Research, 2005.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM* 5(7), 1962, pp. 394–397.

[DM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 4144, pp. 81–94. Springer, 2006. DOI: 10.1007/11817963_11.

[DMH17] Vikram Dhillon, David Metcalf, and Max Hooper. The DAO hacked. In: *Blockchain Enabled Applications*, pp. 67–78. Springer, 2017. DOI: 10.1007/978-1-4842-3081-7_6.

[Dob19] Mihály Dobos-Kovács. Combining testing and formal verification in automotive software development. Bachelor's thesis. Budapest University of Technology and Economics, 2019.

[Dou17] Jules Dourlens. *SafeMath to protect from overflows.* 2017. URL: https://ethereumdev.io/safemath-protect-overflows/.

[DRZ17] Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic predicate abstraction using variable roles. In: *NASA Formal Methods*, Lecture Notes in Computer Science, vol. 10227, pp. 265–281. Springer, 2017. DOI: 10.1007/978-3-319-57288-8_18.

[Dut14] Bruno Dutertre. Yices 2.2. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_49.

[EM00] Javier Esparza and Stephan Melzer. Verification of safety properties using integer programming: beyond the state equation. *Formal Methods in System Design* 16(2), 2000, pp. 159–189. DOI: 10.1023/A:1008743212620.

[EN94] Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets. *J. Inform. Process. Cybernet.* EIK 30(3), 1994, pp. 143–160.

[Eth18] Ethereum. *Solidity Documentation.* 2018. URL: https://solidity.readthedocs.io/en/v0.4.25/.

[Far16] Rebeka Farkas. Verification of Timed Automata by CEGAR-Based Algorithms. Master's thesis. Budapest University of Technology and Economics, 2016.

[FB18] Rebeka Farkas and Gábor Bergmann. Towards reliable benchmarks of timed automata. In: *Proceedings of the 25th PhD Mini-Symposium*, pp. 20–23. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2018.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20(1), 1991, pp. 23–53. DOI: 10.1007/BF01407931.

[Fer+15] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor M González Suárez. Applying model checking to industrial-sized PLC programs. *IEEE Transactions on Industrial Informatics* 11(6), 2015, pp. 1400–1410. DOI: 10.1109/TII.2015.2489184.

[FGG19] Josselin Feist, Gustavo Greico, and Alex Groce. Slither: a static analysis framework for smart contracts. In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, pp. 8–15. IEEE, 2019. DOI: 10.1109/WETSEB.2019.00008.

[Fla+02] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for Java. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 234–245. ACM, 2002. DOI: 10.1145/512529.512558.

[Flo67] Robert W Floyd. Assigning meanings to programs. In: *Proceedings of Symposia in Applied Mathematics Vol. 19*, pp. 19–32. 1967.

[FM10] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In: *Fields of Logic and Computation*, Lecture Notes in Computer Science, vol. 6300, pp. 277–300. Springer, 2010. doi: 10.1007/978-3-642-15025-8_15.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In: *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer, 2013. doi: 10.1007/978-3-642-37036-6_8.

[GD19] Mitchell J Gerrard and Matthew B Dwyer. ALPACA: a large portfolio-based alternating conditional analysis. In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, pp. 35–38. IEEE, 2019. doi: 10.1109/ICSE-Companion.2019.00032.

[GMS18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In: *Principles of Security and Trust*, Lecture Notes in Computer Science, vol. 10804, pp. 243–269. Springer, 2018. doi: 10.1007/978-3-319-89722-6_10.

[God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In: *Computer-Aided Verification*, Lecture Notes in Computer Science, vol. 531, pp. 176–185. Springer, 1991. doi: 10.1007/BFb0023731.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 1254, pp. 72–83. Springer, 1997. doi: 10.1007/3-540-63166-6_10.

[Gul+08] Bhargav S Gulavani, Supratik Chakraborty, Aditya V Nori, and Sriram K Rajamani. Automatically refining abstract interpretations. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 4963, pp. 443–458. Springer, 2008. doi: 10.1007/978-3-540-78800-3_33.

[Gur+15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn verification framework. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer, 2015. doi: 10.1007/978-3-319-21690-4_20.

[Hen+02] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 58–70. 2002. doi: 10.1145/3236950.3236969.

[Hen+04] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. Abstractions from proofs. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 232–244. 2004. doi: 10.1145/982962.964021.

[Hil+18] Everett Hildenbrandt et al. KEVM: a complete formal semantics of the Ethereum virtual machine. In: *Proceedings of the IEEE 31st Computer Security Foundations Symposium*, pp. 204–217. IEEE, 2018. doi: 10.1109/CSF.2018.00022.

[Hir17] Yoichi Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In: *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, vol. 10323, pp. 520–535. Springer, 2017. doi: 10.1007/978-3-319-70278-0_33.

[HK18] Dominik Harz and William Knottenbelt. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. 2018. URL: http://arxiv.org/abs/1809.09805.

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2), 1968, pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[Hoa69] Charles A R Hoare. An axiomatic basis for computer programming. *Communications of the ACM* 12(10), 1969, pp. 576–580. DOI: 10.1145/363235.363259.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In: *Computational Logic*, vol. 9, pp. 135–214. 2014.

[Jär+12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine* 33(1), 2012, pp. 89–92. DOI: 10.1609/aimag.v33i1.2395.

[JD16] Dejan Jovanović and Bruno Dutertre. Property-directed k-induction. In: *Proceedings of the 2016 Conference on Formal Methods in Computer-Aided Design*, pp. 85–92. IEEE, 2016. DOI: 10.1109/FMCAD.2016.7886665.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys* 41(4), 2009, pp. 1–54. DOI: 10.1145/1592434.1592438.

[Joh+13] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: *Proceedings of the 2013 Conference on Formal Methods in Computer-Aided Design*, pp. 201–209. IEEE, 2013. DOI: 10.1109/FMCAD.2013.6679411.

[Jov17] Dejan Jovanović. Solving nonlinear integer arithmetic with MCSAT. In: *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, vol. 10145, pp. 330–346. Springer, 2017. DOI: 10.1007/978-3-319-52234-0_18.

[Kal+18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: analyzing safety of smart contracts. In: *Network and Distributed Systems Security Symposium*, 2018.

[Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: high-performance language-independent model checking. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_61.

[Kor+12] Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Kai Lampka, Niels Lohmann, Emmanuel Paviot-Adet, Yann Thierry-Mieg, and Harro Wimmel. Report on the model checking contest at Petri nets 2011. In: *Transactions on Petri Nets and Other Models of Concurrency VI*, Lecture Notes in Computer Science, vol. 7400, pp. 169–196. Springer, 2012. DOI: 10.1007/978-3-642-35179-2_8.

[Kor+19] Fabrice Kordon, Michael Leuschel, Jaco van de Pol, and Yann Thierry-Mieg. Software architecture of modern model checkers. In: *Computing and Software Science: State of the Art and Perspectives*, Lecture Notes in Computer Science, vol. 10000, pp. 393–419. Springer, 2019. DOI: 10.1007/978-3-319-91908-9_20.

[Kos82] S Rao Kosaraju. Decidability of reachability in vector addition systems. In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pp. 267–281. ACM, 1982. DOI: 10.1145/800070.802201.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures, Second Edition*. Springer, 2016. doi: 10.1007/978-3-662-50497-0.

[KW11] Daniel Kroening and Georg Weissenbacher. Interpolation-based software verification with Wolverine. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 6806, pp. 573–578. Springer, 2011. doi: 10.1007/978-3-642-22110-1_45.

[Lei08] K Rustan M Leino. This is Boogie 2. 2008.

[Lei10] K Rustan M Leino. Dafny: an automatic program verifier for functional correctness. In: *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, vol. 11247, pp. 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.

[Ler11] Jérôme Leroux. Vector addition system reachability problem: a short self-contained proof. *SIGPLAN Notices* 46(1), 2011, pp. 307–316. doi: 10.1145/1925844.1926421.

[Lip76] Richard J Lipton. *The Reachability Problem Requires Exponential Space*. Tech. rep. Yale University, Dept. of Computer Science, 1976.

[LM05] K Rustan M Leino and Peter Müller. Modular verification of static class invariants. In: *FM 2005: Formal Methods*, Lecture Notes in Computer Science, vol. 3582, pp. 26–42. Springer, 2005. doi: 10.1007/11526841_4.

[LMN15] Martin Leucker, Grigory Markin, and Martin R Neuhäußer. A new refinement strategy for CEGAR-based industrial model checking. In: *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, vol. 9434, pp. 155–170. Springer, 2015. doi: 10.1007/978-3-319-26287-1_10.

[Lou+15] Cláudio Belo Lourenço, Si-Mohamed Lamraoui, Shin Nakajima, and Jorge Sousa Pinto. Studying verification conditions for imperative programs. In: *Proceedings of the 15th International Workshop on Automated Verification of Critical Systems*, Electronic Communications of the EASST, vol. 72, pp. 1–15. EASST, 2015. doi: 10.14279/tuj.eceasst.72.1011.

[Löw17] Stefan Löwe. Effective Approaches to Abstraction Refinement for Automatic Software Verification. PhD thesis. University of Passau, 2017.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer* 1(1-2), 1997, pp. 134–152. doi: 10.1007/s100090050010.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K Lahiri. A solver for reachability modulo theories. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 7358, pp. 427–443. Springer, 2012. doi: 10.1007/978-3-642-31424-7_32.

[LS15] Jerome Leroux and Sylvain Schmitz. Demystifying reachability in vector addition systems. In: *Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 56–67. IEEE, 2015. doi: 10.1109/LICS.2015.16.

[LS19] Jerome Leroux and Sylvain Schmitz. Reachability in vector addition systems is primitive-recursive in fixed dimension. In: *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 1–13. IEEE, 2019. doi: 10.1109/LICS.2019.8785796.

[LSS99] K Rustan M Leino, James B Saxe, and Raymie Stata. Checking Java programs via guarded commands. In: *Proceedings of the Workshop on Object-Oriented Technology*, pp. 110–111. Springer, 1999.

[Luu+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269. ACM, 2016. DOI: 10.1145/2976749.2978309.

[May81] Ernst W Mayr. An algorithm for the general Petri net reachability problem. In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pp. 238–246. ACM, 1981. DOI: 10.1145/800076.802477.

[MB08] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, 2008. DOI: 10.1007/978-3-540-78800-3_24.

[McC62] John McCarthy. Towards a mathematical science of computation. In: *IFIP Congress*, pp. 21–28. 1962.

[MCJ18] Andrew Miller, Zhicheng Cai, and Somesh Jha. Smart contracts and opportunities for formal methods. In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Lecture Notes in Computer Science, vol. 11247, pp. 280–299. Springer, 2018. DOI: 10.1007/978-3-030-03427-6_22.

[McM05] Kenneth L McMillan. Applications of Craig interpolants in model checking. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 3440, pp. 1–12. Springer, 2005. DOI: 10.1007/978-3-540-31980-1_1.

[McM06] Kenneth L McMillan. Lazy abstraction with interpolants. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 4144, pp. 123–136. Springer, 2006. DOI: 10.1007/11817963_14.

[Mey19] Bertrand Meyer. *Soundness and completeness: with precision*. 2019. URL: https://bertrandmeyer.com/2019/04/21/soundness-completeness-precision/.

[ML18] Anastasia Mavridou and Áron Laszka. Tool demonstration: FSolidM for designing secure Ethereum smart contracts. In: *Principles of Security and Trust*, Lecture Notes in Computer Science, vol. 10804, pp. 270–277. Springer, 2018. DOI: 10.1007/978-3-319-89722-6_11.

[Mol+18] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 113–116. ACM, 2018. DOI: 10.1145/3183440.3183489.

[Mol19] Vince Molnár. Extensions and generalization of the saturation algorithm in model checking. PhD thesis. Budapest University of Technology and Economics, 2019.

[Mos+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In: *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535. IEEE, 2001. DOI: 10.1145/378239.379017.

[MRS03] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: from refutation to verification. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 2725, pp. 14–26. Springer, 2003. DOI: 10.1007/978-3-540-45069-6_2.

[MS82] Javier Martínez and Manuel Silva. A simple and fast algorithm to obtain all invariants of a generalised Petri net. In: *Application and Theory of Petri Nets*, Informatik-Fachberichte, vol. 52, pp. 301–310. Springer, 1982. DOI: 10.1007/978-3-642-68353-4_47.

[MS99] João P Marques-Silva and Karem Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5), 1999, pp. 506–521.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: a verification infrastructure for permission-based reasoning. In: *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer, 2016. doi: 10.1007/978-3-662-49122-5_2.

[Mue18] Bernhard Mueller. Smashing Ethereum smart contracts for fun and real profit. In: *Proceedings of the 9th Annual HITB Security Conference*, 2018.

[Mül02] Peter Müller. *Modular specification and verification of object-oriented programs.* Springer, 2002. doi: 10.1007/3-540-45651-1.

[Mur89] Tadao Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE* 77(4), 1989, pp. 541–580. doi: 10.1109/5.24143.

[MV20] Milán Mondok and András Vörös. Abstraction-based model checking of linear temporal properties. In: *Proceedings of the 27th PhD Mini-Symposium*, pp. 29–32. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2020.

[Nak08] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system.* 2008. url: http://www.bitcoin.org/bitcoin.pdf.

[Nik+18] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In: *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 653–663. ACM, 2018.

[NO79] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems* 1(2), 1979, pp. 245–257. doi: 10.1145/357073.357079.

[OHJ20] Gustavo A Oliva, Ahmed E Hassan, and Zhen Ming (Jack) Jiang. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empirical Software Engineering*, 2020. doi: 10.1007/s10664-019-09796-5.

[Pas+94] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M Badia. Petri net analysis using Boolean manipulation. In: *Application and Theory of Petri Nets 1994*, Lecture Notes in Computer Science, vol. 815, pp. 416–435. Springer, 1994. doi: 10.1007/3-540-58152-9_23.

[PCP99] Enric Pastor, Jordi Cortadella, and Marco A Peña. Structural methods to improve the symbolic analysis of Petri nets. In: *Application and Theory of Petri Nets 1999*, Lecture Notes in Computer Science, vol. 1639, pp. 26–45. Springer, 1999. doi: 10.1007/3-540-48745-X_3.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 697, pp. 409–423. Springer, 1993. doi: 10.1007/3-540-56922-7_34.

[Pet62] Carl Adam Petri. Kommunikation mit Automaten. PhD thesis. Technische Universität Darmstadt, 1962.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[Poe97] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis. Technical University of Munich, 1997.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In: *International Symposium on Programming*, Lecture Notes in Computer Science, vol. 137, pp. 337–351. Springer, 1982. DOI: 10.1007/3-540-11494-7_22.

[RE14] Zvonimir Rakamarić and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 8559, pp. 106–113. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_7.

[Rey+17] Andrew Reynolds, Cesare Tinelli, Dejan Jovanović, and Clark Barrett. Designing theory solvers with extensions. In: *Frontiers of Combining Systems*, Lecture Notes in Computer Science, vol. 10483, pp. 22–40. Springer, 2017. DOI: 10.1007/978-3-319-66167-4_2.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society* 74(2), 1953, pp. 358–366.

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79(6), 2010, pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

[RW19] Cedric Richter and Heike Wehrheim. PeSCo: predicting sequential combinations of verifiers. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 11429, pp. 229–233. Springer, 2019. DOI: 10.1007/978-3-030-17502-3_19.

[Sal16] Gyula Sallai. Development of a Verification Compiler for C Programs. Bachelor's Thesis. Budapest University of Technology and Economics, 2016.

[Sal19] Gyula Sallai. LLVM IR-based Transformations for Software Model Checking. Master's thesis. Budapest University of Technology and Economics, 2019.

[Sch02] Philippe Schnoebelen. The complexity of temporal logic model checking. *Advances in modal logic* 4(35), 2002, pp. 393–436.

[Sch03] Karsten Schmidt. Using Petri net invariants in state space construction. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 2619, pp. 473–488. Springer, 2003. DOI: 10.1007/3-540-36577-X_35.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., 1986.

[Sch99] Karsteb Schmidt. Stubborn sets for standard properties. In: *Application and Theory of Petri Nets*, Lecture Notes in Computer Science, vol. 1639, pp. 46–65. Springer, 1999. DOI: 10.1007/3-540-48745-X_4.

[Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation* 3, 2007, pp. 141–224.

[Ser+19] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages* 3(OOPSLA), 2019, 185:1–185:30. DOI: 10.1145/3360611.

[SFB07] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pp. 112–122. ACM, 2007. DOI: 10.1145/1250734.1250748.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In: *Proceedings of the 2000 Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, vol. 1954, pp. 127–144. Springer, 2000. doi: 10.1007/3-540-40922-X_8.

[ST17] Gyula Sallai and Tamás Tóth. Boosting software verification with compiler optimizations. In: *Proceedings of the 24th PhD Mini-Symposium*, pp. 66–69. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2017. doi: 10.5281/zenodo.291903.

[Sza94] Nick Szabo. *Smart contracts.* 1994.

[Teg18] Tamás Tegzes. Applying Incremental, Inductive Model Checking to Software. Bachelor's thesis. Budapest University of Technology and Economics, 2018.

[TM17] Tamás Tóth and István Majzik. Lazy reachability checking for timed automata using interpolants. In: *Formal Modelling and Analysis of Timed Systems*, Lecture Notes in Computer Science, vol. 10419, pp. 264–280. Springer, 2017. doi: 10.1007/978-3-319-65765-3_15.

[TM18] Tamás Tóth and István Majzik. Lazy reachability checking for timed automata with discrete variables. In: *Model Checking Software*, Lecture Notes in Computer Science, vol. 10869, pp. 235–254. Springer, 2018. doi: 10.1007/978-3-319-94111-0_14.

[Tsa+18] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: practical security analysis of smart contracts. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 67–82. ACM, 2018. doi: 10.1145/3243734.3243780.

[TTC96] Marco Tilgner, Yukio Takahashi, and Gianfranco Ciardo. SNS 1.0: synchronized network solver. In: *Proceedings of the 1st International Workshop on Manufacturing and Petri Nets*, pp. 215–234. 1996.

[Tul+14] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V Nori. MUX: algorithm selection for software model checkers. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 132–141. ACM, 2014. doi: 10.1145/2597073.2597080.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math* 58, 1936, pp. 345–363.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In: *Advances in Petri Nets 1990*, Lecture Notes in Computer Science, vol. 483, pp. 491–515. Springer, 1991. doi: 10.1007/3-540-53863-1_36.

[VDB11] András Vörös, Dániel Darvas, and Tamás Bartha. Bounded saturation based CTL model checking. In: *Proceedings of the 12th Symposium on Programming Languages and Software Tools, SPLST'11*, pp. 149–160. Tallinn University of Technology, Institute of Cybernetics, 2011.

[VG09] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In: *Proceedings of the 2009 Conference on Formal Methods in Computer-Aided Design*, pp. 1–8. 2009. doi: 10.1109/FMCAD.2009.5351148.

[Vör18] András Vörös. Symbolic Verification of Petri Net Based Models. PhD thesis. Budapest University of Technology and Economics, 2018.

[VWM15] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE* 103(11), 2015, pp. 2021–2035.

[Wan+20] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in Azure blockchain. In: *Verified Software. Theories, Tools, and Experiments*, Lecture Notes in Computer Science, vol. 12031, pp. 87–106. Springer, 2020. DOI: 10.1007/978-3-030-41600-3_7.

[WBK20] Lukas Westhofen, Philipp Berger, and Joost-Pieter Katoen. Benchmarking software model checkers on automotive code. In: *NASA Formal Methods*, 2020. (In press).

[Web+19] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015–2018. *Journal on Satisfiability, Boolean Modeling and Computation* 11(1), 2019, pp. 221–259.

[Wei81] Mark Weiser. Program slicing. In: *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449. IEEE, 1981. DOI: 10.1109/TSE.1984.5010248.

[WG16] Hadley Wickham and Garrett Grolemund. *R for data science: import, tidy, transform, visualize, and model data.* O'Reilly Media, Inc., 2016.

[Woh+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, 2012. DOI: 10.1007/978-3-642-29044-2.

[Wol18] Karsten Wolf. Petri net model checking with LoLA 2. In: *Application and Theory of Petri Nets and Concurrency*, Lecture Notes in Computer Science, vol. 10877, pp. 351–362. Springer, 2018. DOI: 10.1007/978-3-319-91268-4_18.

[Wol19] Karsten Wolf. How Petri net theory serves Petri net model checking: a survey. In: *Transactions on Petri Nets and Other Models of Concurrency XIV*, Lecture Notes in Computer Science, vol. 11790, pp. 36–63. Springer, 2019. DOI: 10.1007/978-3-662-60651-3_2.

[Woo17] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger.* 2017. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[WW11] Harro Wimmel and Karsten Wolf. Applying CEGAR to the Petri net state equation. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 6605, pp. 224–238. Springer, 2011. DOI: 10.1007/978-3-642-19835-9_19.

[WW12] Harro Wimmel and Karsten Wolf. Applying CEGAR to the Petri net state equation. *Logical Methods in Computer Science* 8(3), 2012, pp. 1–15.