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. Thesis Booklet

Ákos Hajdu

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

Budapest
2020

1 Preliminaries and Objectives

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 serious 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 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 practical size. Nevertheless, the promise of formal correctness guarantees has spawned a great interest and a wide variety of approaches have been developed, including symbolic methods [Bur+90], partial order reduction techniques [Val91; God91; Pel93], bounded model checking [Bie+99], abstraction [CGL94; Cla+03] and modular verification [Mül02]. 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*.

1.1 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.

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.

		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.

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*.

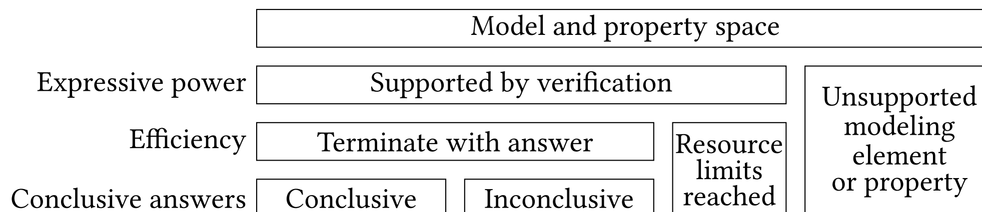

Figure 2. Possible outcomes of verification in practice. When verification cannot give a conclusive answer, it can terminate inconclusively, reach its resource limits or encounter an unsupported feature.

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.[1]

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

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

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.

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.

1.2 Overview

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.

1.3 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

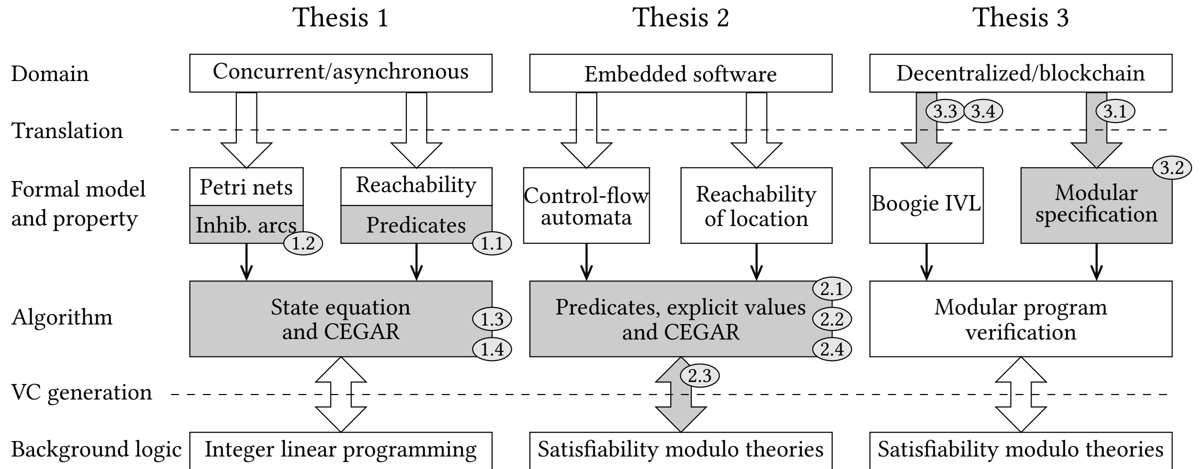

Figure 3. Overview of the problem domains and the verification approaches used in each thesis. Own contributions are denoted with a filled background.

	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.

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.

1.4 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 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 [BLW15; 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.

1.5 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.

2 New Results

New results are grouped by the three main domains (concurrent and asynchronous systems, embedded software code and blockchain-based decentralized systems). Most of the research has been carried out in collaboration with other researchers, but I emphasize my own contributions.

2.1 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[2] 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]. My contributions are summarized as follows.

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

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].

2.2 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[3] 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. My contributions are summarized as follows.

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

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].

2.3 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[4] 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. My contributions are summarized as follows.

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

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

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.

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.[5] 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.[6] Furthermore, a US patent including (but not limited to) my results was also filed in December 2018 and is currently pending.

3 Application of the New Results

3.1 Extensions to the CEGAR Approach on Petri Nets

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*[7] 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].

3.2 Efficient Strategies for CEGAR-based Software Model Checking

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[8] 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.

[5] https://solidity-summit.ethereum.org/
[6] https://fscd-ijcar-2020.org/workshops#SMT
[7] http://www.evopro.hu/en
[8] http://cern.ch/plcverif/

3.3 Modular Specification and Verification of Smart Contracts

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].

4 Publication List

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

4.1 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.

4.2 Additional Publications (Not Linked to Theses)

International Conference and Workshop Papers

[c14] Ákos Hajdu, Róbert Német, 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.

4.3 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).

References

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[BLW15] 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.

[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.

[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.

[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.

[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.

[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+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.

[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.

[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.

[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.

[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.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[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.

[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.

[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.

[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.

[Flo67] Robert W Floyd. Assigning meanings to programs. In: *Proceedings of Symposia in Applied Mathematics Vol. 19*, pp. 19–32. 1967.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[McC62] John McCarthy. Towards a mathematical science of computation. In: *IFIP Congress*, pp. 21–28. 1962.

[Mey19] Bertrand Meyer. *Soundness and completeness: with precision.* 2019. URL: https://bertrandmeyer.com/2019/04/21/soundness-completeness-precision/.

[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.

[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.

[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.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[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.

[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.

[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.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., 1986.

[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.

[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.

[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.

[Vör18] András Vörös. Symbolic Verification of Petri Net Based Models. PhD thesis. Budapest University of Technology and Economics, 2018.

[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.