JOHANNES KEPLER
UNIVERSITÄT LINZ

Submitted by
Nils Froleyks

Submitted at
Institute for Formal Models and Verification

First Supervisor
Prof. Dr. Armin Biere

Second Supervisor
Univ.-Prof. Dr. Martina Seidl

July 2025

Deep Integration of SAT Solving and Model Checking

Doctoral Thesis

to obtain the academic degree of

Doktor der Technischen Wissenschaften

in the Doctoral Program

Technischen Wissenschaften

JOHANNES KEPLER UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Once just considered the archetypal intractable problem, Boolean satisfiability (SAT) has flourished into one of the most effective tools for automated reasoning. One of its most successful applications is model checking, where SAT solvers are used to verify the correctness of systems by checking whether certain properties hold. This thesis investigates how to deepen the integration between SAT solving and model checking to improve performance and increase trust in the results.

We begin by presenting the state-of-the-art incremental SAT solver CaDiCaL 2.0. We extend its assumption-based interface with clause assumptions, and demonstrate how this enhancement accelerates IC3-based model checking. Building on CaDiCaL, we introduce a new backbone extraction tool, CadiBack. A formula's backbone—the set of literals true in every satisfying assignment—can benefit various SAT-based applications, including verification, state space approximation, and fault localization. As part of this work, we present a polynomial-time backbone extraction algorithm and evaluate it as a preprocessor for CadiBack. Additionally, we formalize ternary simulation through the lens of abstract interpretation and propose a technique to improve accuracy using backbone extraction.

We then address how to achieve a similar level of trust in model checking as is commonly vested in SAT solvers. To this end, we develop a certification approach for hardware model checking. Specifically, we present certificate constructions for the preprocessing techniques of phase abstraction and constraint extraction. Our efforts culminate in the Hardware Model Checking Competition 2024, which marks the introduction of mandatory certificates for all participating tools. The results convincingly show the practical efficiency and effectiveness of our approach.

Finally, this thesis further explores competitions as a scientific method for advancing solver technology. Drawing on the author's experience co-organizing several SAT and hardware model checking competitions, we examine what insights into solver innovation can be gained from these events.

Zusammenfassung

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) galt zunächst als Paradebeispiel für nicht effizient lösbare Probleme. Seitdem hat es sich zu einem der wichtigsten Werkzeuge der automatisierten Beweisführung entwickelt. Es bildet die Grundlage für eine Vielzahl von Anwendungen – insbesondere im Model Checking, wo SAT-Solver eingesetzt werden, um die Korrektheit von Modelleigenschaften zu beweisen.

Diese Dissertation untersucht, wie SAT-Solving und Model Checking weiter integriert werden können, um effizientere automatisierte Verifikationsverfahren zu ermöglichen und einen höheren Grad an Vertrauen in ihre Korrektheit zu erlangen.

Zu Beginn stellen wir den modernen inkrementellen SAT-Solver CaDiCaL 2.0 vor. Wir erweitern dessen *annahmenbasierte* Schnittstelle für inkrementelle Anwendungen um Klausel-Annahmen und zeigen, dass diese Erweiterung IC3-basiertes Model Checking signifikant beschleunigt. Aufbauend auf CaDiCaL entwickeln wir die Anwendung CadiBack zur Backbone-Extraktion. Der Backbone einer Formel – die Menge aller Literale, die in jeder erfüllenden Belegung wahr sind – kann verschiedene SAT-basierte Anwendungen unterstützen, etwa Verifikation, Zustandsraumapproximation und Fehlereingrenzung. Im Rahmen dieser Arbeit stellen wir einen polynomiellen Algorithmus zur Backbone-Extraktion vor und evaluieren ihn als Vorverarbeitungsschritt für CadiBack. Darüber hinaus formalisieren wir ternäre Simulation im Rahmen abstrakter Interpretation und präsentieren eine Technik zur Genauigkeitssteigerung, die auf Backbone-Informationen basiert.

Anschließend widmen wir uns der Frage, wie sich in der Modellprüfung ein ähnliches Maß an Vertrauen erreichen lässt wie bei SAT-Solvern. Zu diesem Zweck entwickeln wir einen Zertifizierungsansatz für die Verifikation von Hardware-Systemen. Konkret präsentieren wir Zertifikatskonstruktionen für die Vorverarbeitungstechniken der Phasenabstraktion und Constraint-Extraktion. Unsere Arbeit kulminiert in der Hardware Model Checking Competition 2024, in der erstmals für alle teilnehmenden Modellprüfer Zertifikate verpflichtend eingeführt wurden. Die Ergebnisse belegen eindrucksvoll die praktische Effizienz und Wirksamkeit unseres Ansatzes.

Abschließend untersuchen wir, welchen Beitrag Wettbewerbe als wissenschaftliche Methode zur Weiterentwicklung von Solver-Technologien leisten können.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Armin, for his unwavering support, guidance, and encouragement throughout this journey. His mentorship has been invaluable, and he serves as both an inspiration and a role model.

I am also sincerely grateful to my second advisor, Martina, for her steady support.

This thesis would not have been possible without the contributions of my collaborators, to whom I extend my heartfelt thanks.

Finally, I am profoundly thankful to my family. To my wife, whose support and understanding have been a cornerstone of this endeavor—thank you.
To our child, who has not made it too difficult.

Contents

1 Introduction **1**
1.1 Outline and Contributions 3

2 Background **5**
2.1 SAT Solving 5
2.2 SAT Solving Certificates 6
2.3 Model Checking 7
2.4 Model Checking Certificates 10

3 Discussion on Published Work **13**

4 CaDiCaL 2.0 **17**
4.1 Introduction 18
4.2 Architecture 20
4.3 External Propagator 21
4.4 Proofs 22
4.5 Tracer Interface 22
4.6 Constraints and Flipping 23
4.7 Interpolation 24
4.8 Testing and Debugging 24
4.9 Experiments 25
4.10 Conclusion 27
4.11 Appendix 27

5 Single Clause Assumption without Activation Literals to Speed-up IC3 **35**
5.1 Introduction 36
5.2 Incremental SAT and IC3 38
5.3 Assuming Clauses 38
5.4 Experiments 39
5.5 Conclusion 43

6 CadiBack: Extracting Backbones with CaDiCaL **45**
6.1 Introduction 46
6.2 Basic Concepts and Notations 47
6.3 Algorithms in MINIBONES 48
6.4 Improved Algorithms in CADIBACK 49
6.5 Implementation 52

6.6 Experiments 53
6.7 Conclusion 55

7 BIG Backbones 61
7.1 Introduction 62
7.2 Preliminaries 62
7.3 Single Unit Resolution Backbone 63
7.4 BIG Backbones 65
7.5 Related Techniques 67
7.6 Implementation and Evaluation 68
7.7 Conclusion 69

8 Ternary Simulation as Abstract Interpretation 71
8.1 Introduction 72
8.2 Circuit 73
8.3 Ternary Simulation as Abstract Interpretation 74
8.4 Widening 76
8.5 Narrowing 76
8.6 Termination via Subsumption 77
8.7 Evaluation 77

9 Certifying Phase Abstraction 79
9.1 Introduction 80
9.2 Background 81
9.3 Periodic Signals 82
9.4 Extending Phase Abstraction 83
9.5 Certification 85
9.6 Implementation 91
9.7 Experimental Evaluation 92
9.8 Conclusion 95
9.9 Appendix 96

10 Certifying Constraints in Hardware Model Checking 101
10.1 Introduction 102
10.2 Constrained circuits 104
10.3 Certifying Constraint Extraction 106
10.4 Relaxed simulation 110
10.5 Extending Circuits with Oracles 112
10.6 Experimental Evaluation 116
10.7 Conclusion 117

11 Introducing Certificates to the Hardware Model Checking Competition 119
11.1 Introduction 120
11.2 Related Work 122

11.3 Certificate Format 122
11.4 Evaluation 126
11.5 Conclusion 131
11.6 Appendix 131

12 Conclusion and Future Work 135

Bibliography 139

Chapter 1

Introduction

Modern software and hardware systems are marvels of complexity, often comprising millions of lines of code or intricate circuit designs. Yet this complexity comes at a cost: undetected errors can lead to catastrophic failures. A striking example is the Pentium FDIV bug, uncovered by Thomas R. Nicely in 1994, which caused Intel Pentium processors to produce incorrect floating-point calculations—a flaw that undermined trust in otherwise groundbreaking technology. Such incidents underscore a critical imperative: ensuring system correctness *prior to deployment* is not merely a technical challenge, but a societal necessity.

Formal methods offer a rigorous solution, providing automated reasoning techniques to verify that software and hardware systems meet their specified requirements. Among these, model checking [33] stands out as a powerful technique, systematically exploring whether a system model satisfies a logical specification. At the heart of many model checkers lies the propositional satisfiability (SAT) solver—a computational engine that determines whether a Boolean formula can be satisfied. Modern SAT solvers, powered by Conflict-Driven Clause Learning (CDCL), are indispensable in state-of-the-art hardware verification, where model checkers repeatedly invoke them to answer complex queries. This thesis explores the profound interplay between SAT solving and model checking, aiming to push the boundaries of both fields.

One key enabler in this integration is *incremental SAT solving*. Techniques like bounded model checking [20], IC3 [34], and k-induction [35] rely on solving sequences of closely related SAT instances. By leveraging incremental solver interfaces, these methods minimize redundant computation and enhance efficiency. Yet the demands of model checking expose limitations in traditional SAT approaches, spurring the development of new techniques that may transcend their original scope. Circuit-specific SAT, for example, exploits structural insights from hardware designs to boost performance [1, 2]. This leads to our first research question:

RQ1: *How can we optimize SAT solving for aberrant applications?*

We address this by exploring incremental strategies and circuit-aware techniques, seeking not only performance gains but also simplified solver interaction.

Beyond efficiency, SAT solvers offer a unique strength: the ability to produce *proofs*. For satisfiable instances, they provide witnesses in the form of satisfying assignments; for unsatisfiable instances, resolution proofs. In SAT competitions, solvers are required to produce such certificates, which fosters trust and robustness. This capability motivates our second question:

RQ2: *How can model checking reach the same level of trust as SAT solving?*

By developing certificates for model checking, we aim for end-to-end trustworthiness—ensuring that every verification step is proven correct.

Competitions have long fueled progress in automated reasoning. Events such as the SAT Competition and the Hardware Model Checking Competition provide benchmarks, compare solvers on even ground, and ignite innovation. As an organizer of several such events, the author of this thesis has witnessed how important competitions are to any field of research. This motivates our third question:

RQ3: *What is the role of competitions in scientific inquiry?*

Through contributions to and analysis of these events, this thesis highlights their influence and importance as proving grounds for ideas and as an aspect of the scientific method.

This thesis addresses these questions through a multifaceted approach, with a particular emphasis on hardware verification. We begin by presenting CaDiCaL, a state-of-the-art SAT solver designed with a rich library interface for incremental solving. Despite its extensive feature set, experimental results show that CaDiCaL achieves competitive performance in both stand-alone and incremental settings. More importantly, as of writing, it is the only solver that fully supports LRAT proof generation, enabling unparalleled efficiency in proof checking. We leverage this and CaDiCaL's circuit-specific optimizations when validating model checking certificates with a reduced base of trust. We further extend the traditional assumption-based interface with clause assumptions and use this new extension to accelerate the IC3 model checking algorithm.

Building on this line of research, we developed CadiBack, a backbone extraction tool that leverages CaDiCaL's unique features. We extend existing algorithms with polynomial-time backbone extraction from the binary implication graph. We use the backbone extraction technique we developed to increase the precision of cube simulation—a generalization of ternary simulation—which we propose and formalize as an instance of abstract interpretation. Cube simulation serves as a core component in several preprocessing techniques for hardware model checking, most notably Temporal Decomposition and Phase Abstraction. We further formalize—and, in some cases, generalize—these techniques with the end goal of developing complete certificate constructions. In other words, we show how a model checker that employs such preprocessing techniques can generate certificates to prove the correctness of the entire model checking process, including preprocessing. We present a strong argument for the practical viability of our approach by introducing certificates to the Hardware Model Checking Competition and presenting a detailed evaluation focused on the produced certificates.

1.1 Outline and Contributions

This thesis is structured as a cumulative dissertation and is divided into three main parts. The first part, presented in this chapter, provides a high-level overview of the contributions, followed by essential background material in the next chapter. The second part consists of Chapters 4–11, which include eight peer-reviewed papers. Finally, the thesis concludes with a discussion of future research directions.

It is important to acknowledge that the work presented in this thesis would not have been possible without the support and collaboration of my co-authors. Below, I outline my specific contributions to each paper.

Chapter 4 *CaDiCaL 2.0* [4] with Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury and Florian Pollitt. In Proceedings of Computer Aided Verification—36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024.

The author implemented the constraint feature (further described in Chapter 5) in CaDiCaL. He attends the regular development meetings and provides discussion on applications, as well as evaluation and presentation of experimental results. In this paper, he worked on the new constraint and flipping features, and the overall presentation of the presented components. He was responsible for conducting all the evaluation presented in the paper with the exception of the interpolation run.

Chapter 5. *Single Clause Assumption without Activation Literals to Speed-up IC3* [5] with Armin Biere. In Proceedings of Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19–22, 2021.

N. Froleyks is the **first author** of the paper. The initial motivation was provided by A. Biere, and the algorithm emerged from discussions between the author and A. Biere. N. Froleyks handled both the implementation and the evaluation.

Chapter 6. *CadiBack: Extracting Backbones with CaDiCaL* [6] with Wenxi Wang and Armin Biere. In Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4–8, 2023, Alghero, Italy.

A. Biere implemented the tool described in the paper. The experimental evaluation was carried out jointly by A. Biere and N. Froleyks. The author reviewed the code and described the algorithms in the paper.

Chapter 7. *BIG Backbones* [7] with Emily Yu and Armin Biere. In Proceedings of Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24–27, 2023.

N. Froleyks is the **first author**. He implemented the extensions to CadiBack, developed the theoretical results, and performed the evaluation. The KB3 algorithm is an adaptation of an idea originally implemented by A. Biere in a different context.

Chapter 8. *Ternary Simulation as Abstract Interpretation* [8] with Emily Yu and Armin Biere. In Proceedings of MBMV 2024; 27. Workshop.

N. Froleyks is the **first author** of this workshop paper. He developed the formalization and implemented the narrowing and widening operators as well as the new termination strategy. The author also conducted the experimental evaluation.

Chapter 9. *Certifying Phase Abstraction* [9] with Emily Yu, Armin Biere, Keijo Heljanko. In Proceedings of Automated Reasoning—12th International Joint Conference, IJCAR 2024, Nancy, France, July 3–6, 2024.

N. Froleyks is the **first author**. The original certificate generation for phase abstraction was developed by E. Yu. The author extended the phase abstraction technique and the certification to cover it. He further implemented both the preprocessing technique and its certificate construction into his model checker. He was responsible for the evaluation presented in the paper.

Chapter 10. *Certifying Constraints in Hardware Model Checking* [10] with Emily Yu, Armin Biere, and Keijo Heljanko. Submitted.

The author worked on the certificate constructions detailed in the paper and contributed the completeness proofs. He devised the weak induction check. All presented tools and experiments were developed by the author.

Chapter 11. *Introducing Certificates to the Hardware Model Checking Competition* [11] with Emily Yu, Mathias Preiner, Armin Biere, and Keijo Heljanko. In Proceedings of Computer Aided Verification–37th International Conference, CAV 2025, Zagreb, Croatia, July 21–25, 2025.

N. Froleyks is the **first author** of the paper. In collaboration with E. Yu, the author extended the theory of hardware model checking certificates for the version used in the competition. He implemented this theory in the checker deployed for the competition, performed the evaluation, and contributed the correctness proofs in the paper.

In addition to the papers included in this thesis, I contributed to several other works [1–3, 12–16], that are not part of this dissertation. Furthermore, I co-authored papers [17–19] partially based on earlier master's thesis work, which have also been excluded.

Beyond the Hardware Model Checking Competition 2024 [11] discussed in this thesis, I actively contributed to the competitions in our community by organizing [20–23], submitting benchmarks [24–27], and joining [28, 29].

Chapter 2

Background

This chapter provides a high-level overview of the foundational concepts relevant to this thesis. While Chapters 4–11 present self-contained, peer-reviewed publications, this chapter serves as a unifying introduction to the broader context. We begin by introducing satisfiability solving and model checking, which form the core theoretical and practical underpinnings of the work presented in this dissertation.

2.1 SAT Solving

We provide a brief introduction to the Boolean satisfiability (SAT) problem. The standard logical connectives are denoted by the symbols $\neg$ (negation), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence), $\wedge$ (conjunction), and $\vee$ (disjunction).

- A *literal* is either a Boolean variable or its negation.
- A *clause* is a set of literals combined by disjunction.
- A formula in *conjunctive normal form (CNF)* is a conjunction of clauses.
- An *assignment* $\sigma : V \rightharpoonup \bot, \top$ is a partial function mapping variables to truth values $\bot, \top$ (false, true). Extending an assignment means assigning additional variables without changing the truth values of already assigned ones.

Further, $\texttt{VAR}(F)$ denotes the variables of a formula F, i.e., $\ell \in \texttt{VAR}(F)$ if either ℓ or $\neg\ell$ appears in one of the clauses. A formula under an assignment evaluates to the result of basic simplification after replacing positive literals with their assigned truth values and negative literals with the negation of the assigned values. If the assignment is total—i.e., every variable is assigned—the formula evaluates to either true or false.

The satisfiability problem asks whether there exists an assignment for the variables of a given formula such that the formula evaluates to true. If at least one such assignment exists, the formula is satisfiable (SAT); otherwise, it is unsatisfiable (UNSAT). A satisfying assignment to all variables is also called a *model* of the formula.

When we encode some statement as a Boolean formula that evaluates to true exactly when the statement holds, we also refer to such a formula as a *predicate*. Conversely, when no particular meaning is attached to the truth value of a formula, we also refer to it as a Boolean *function*.

Example 2.1. Consider the Boolean formula $\phi = (a \to \neg b) \wedge (a \to b)$. A satisfying assignment for this formula is $a \mapsto \bot, b \mapsto \top$. Therefore, ϕ is satisfiable. In contrast, consider $\varphi = a \wedge (a \to \neg b) \wedge (a \to b)$. No assignment satisfies φ, so it is unsatisfiable.

The formulas in the example above are not in CNF. Any Boolean formula can be converted into CNF by eliminating implications ($\to$) and applying Tseitin transformation [36]. Since $A \to B$ is equivalent to $\neg A \vee B$, implications can be systematically eliminated to facilitate CNF conversion. Equivalences ($\leftrightarrow$) can similarly be translated into two clauses.

State-of-the-art SAT solvers employ conflict-driven clause learning (CDCL) [37] in combination with *inprocessing*. The CDCL algorithm guesses sensible assignments to individual variables, considers their consequences, and if a conflict is found, learns a clause that prevents the same from happening again. The solver either eventually derives the empty clause (proving unsatisfiability) or successfully assigns all variables without conflict (yielding a model). Inprocessing is interleaved with the search and transforms the formula into an equisatisfiable one that is hopefully easier to solve. Further details are presented in Chapter 4.

Besides solving individual formulas, *incremental* SAT solving allows the user to efficiently solve a sequence of related SAT problems. For many problems it is necessary to use information from a previous query to the solver to slightly extend the formula. This is especially true for problems which are hard for a complexity class exceeding NP or coNP, such as the PSPACE-complete model checking problem. See Chapter 5 for an introduction to incremental SAT solving.

Another task that builds on SAT solving is *backbone extraction*, which asks not only whether a satisfying assignment exists, but also which literals are true in *every* such assignment. We discuss backbone extraction in detail in Chapters 6 and 7.

2.2 SAT Solving Certificates

SAT solvers are widely used in applications where correctness is critical. Together with the fact that SAT solvers are complicated pieces of software, the need arises to convince users that their results are trustworthy.

Certification essentially means providing a witness that reduces the complexity of a problem to the point where a verifier can be convinced that it has been solved. For satisfiable CNF formulas, this is a simple task: the model serves as the certificate and can be checked by going over the formula clause by clause. This verification is simple to implement and runs in linear time, in contrast to the NP-hardness of the satisfiability problem itself. If the formula is unsatisfiable, the task becomes significantly harder as outlined below.

When a SAT solver determines unsatisfiability, it generates a deductive proof for the correctness of this result. These proofs can then be checked by independent tools known as proof checkers. To facilitate this process, it is important to establish a fixed format which is expressive enough to describe the techniques employed by the SAT solver, and at the same time simple to check. Proof checkers fall into two broad categories:

verified and unverified, or colloquially: correct and fast. Verified proof checkers are developed within interactive theorem provers such as Isabelle [38] or in formally verified language implementations such as CakeML [39]. They provide strong guarantees of correctness but are complex to implement and difficult to optimize for performance. On the other hand, unverified proof checkers are comparatively simple C programs that rely on the relative simplicity of the proof checking task to increase trust in the certified result. Additionally, the intuition that an incorrect proof checker would only accept an independently produced buggy proof if the flaws in both align precisely results in a kind of "quadratic improvement in trust".

A number of different proof formats have been established. They differ in how expressive they are, and often come in a variant with and without annotations to make proof checking easier. At the lower end of expressiveness we have DRUP which combines *reverse unit propagation* (RUP) [40] with the deletion of clauses. In this case, the aforementioned annotations are the clauses which are involved in the propagation. If they are included the format is called LRUP and proof checking is linear-time. Judging by the winners of the SAT competition [14], all practically relevant SAT-solving techniques can be expressed using RUP. That is until the competition in 2023, where a rather rudimentary combination of the SAT solver CaDiCaL with bounded variable addition (BVA) [41] won the competition. Since then, both CaDiCaL and Kissat—two of the strongest SAT solvers developed over the past decade—have implemented BVA. This is noteworthy as the impressively efficient and verified proof checker lrat_isa [38] only supports LRUP and thus can no longer be used with CaDiCaL when BVA (enabled by default via `--factor`) is active.

The long-standing standard format is DRAT [42], with its linear variant LRAT [43], implemented in the widely used unverified tools DRAT-trim and LRAT-trim. Moving further up in expressiveness we have propagation redundancy (PR) [44], supported by the verified tool cake_lpr [39]. Lastly, VeriPB [45] extends to the pseudo-Boolean domain and further features strong proof rules [46].

2.3 Model Checking

Model checking verifies whether the model of a system satisfies a given property.[1] In this thesis, we focus on *safety* properties.

Example 2.2. Consider a simple transition system modeled using two Boolean variables a and b. The initial state is state 0, encoded symbolically as $\neg a \wedge \neg b$. We are interested in verifying the property $\neg a \vee \neg b$, which is violated only by state 3 (i.e., when $a \wedge b$ holds). However, since state 3 is not reachable from the initial state, the model is *safe*.

$$\begin{tikzcd}
\text{start} \arrow[r] & 0 \arrow[loop above] \arrow[r] & 1 \arrow[r] & 2 \arrow[loop above]
\end{tikzcd}$$

[1]Note that the term *model* here is not the same as a satisfying assignment for SAT formulas. In the context of neural networks, the term has yet another meaning.

A transition system is typically defined as a tuple $M = (V, I, T, P)$, where:

- V is a finite set of state variables;
- $I(V)$ a formula over V encoding the set of initial states;
- $T(V, V')$ a formula encoding the transition relation, where we use the primed variables V' to denote the next-state variables;
- $P(V)$ a formula encoding the set of good states, where the property holds.

Safety properties such as P are characterized by having *finite counter examples* — sequences of states from an initial state to a state violating P, also called bad traces. Such a counterexample corresponds to a model of the formula (for some k):

$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \bigvee_{i \in [0,k]} \neg P(s_i).$$

Solving this formula for increasing values of k using a SAT solver forms the core of the well-known *Bounded Model Checking (BMC)* algorithm [47].

These counterexamples can be viewed as certificates for unsafe instances of the model checking problem, analogous to satisfying assignments for SAT formulas. However, unlike satisfying assignments, counterexamples may be exponentially large—up to $2^{|V|}$ states.

To prove that a model checking problem is safe using bounded model checking the formula above would need to be shown unsatisfiable for $k = 2^{|V|}$. This is infeasible for all but the smallest systems which is why bounded model checking is usually considered incomplete, meaning that it is only used to find counterexamples, not to prove their absence.

However, other model checking techniques such as k-induction [48] and IC3 [49] [2] are complete and can generate certificates for safe model checking instances.

2.3.1 Circuits

In Chapters 8, 9, 10, and 11, we use Boolean circuits—rather than transition systems—to describe models.

The main motivation for using the circuit formalism described below, is that it closely aligns with the AIGER [50, 51] format used in the Hardware Model Checking Competition [20] and in industrial practice.

In principle, any system that can be modeled as a transition system can also be expressed as a Boolean circuit, however, they are particularly well-suited for describing hardware designs, such as CPUs. They further come with a number of structural restrictions natural for such designs, which we will use in Chapter 9 to generate smaller certificates and check them more efficiently.

A sequential Boolean circuit is defined as a tuple $M = (I, L, R, F, P, C)$, where:

[2] See also Chapter 5.

- **Inputs I.** A finite set of primary input variables modeling the environment. These are the only source of nondeterminism and can be treated as randomly assigned at each step. A subtlety is that in many visualizations (e.g., Chapters 5, 9, 10), only latch values are depicted, while input variations are shown via multiple transition edges per state. In these examples, the safety property P typically depends only on the latch variables L to avoid confusion.

- **Latches L.** A finite set of state-holding variables. A state of the circuit is defined by an assignment to inputs and latches. We collectively refer to inputs, latches, and their negations as the *literals* of a circuit.

- **Reset Functions R.** Every latch ℓ is associated with a reset function r_ℓ in R. The r_ℓ are Boolean functions over I and L, so have a fixed value for a given state. The predicate
$$R\{L\} = \bigwedge_{\ell \in L} \ell \leftrightarrow r_\ell$$
holds if all latches take their reset values, i.e., the state is an initial (reset) state.

- **Transition Functions F.** Each latch $\ell \in L$ has a transition function f_ℓ, determining its next value based on I and L. The transition predicate
$$F_{0,1}\{L\} = \bigwedge_{\ell \in L} (\ell_1 \leftrightarrow f_\ell(I_0, L_0)),$$
references two successive states. We adopt the convention of indexing formulas (e.g., R_0 or $F_{0,1}$) to refer to specific time steps without explicitly indexing variables.

  In AIGER, reset and transition functions are encoded using AND gates and negations. We abstract away from this representation except in Chapter 8, which focuses on ternary simulation and relies on the circuit's structure.

- **Property P.** A Boolean predicate over the circuit state characterizing the set of *good states* (i.e., a safety property).

- **Constraint C.** A Boolean predicate that restricts the set of admissible states, typically used to model environment assumptions. Many models do not have a constraint ($C = \top$), which has useful consequences: every state has at least one successor, and every finite trace can be extended to an infinite one.

A *bad trace* in a circuit corresponds to:
$$R_0 \;\wedge\; \bigwedge_{i \in [0,n)} F_{i,i+1} \;\wedge\; \bigwedge_{i \in [0,n]} C_i \;\wedge\; \neg P_n.$$

In the above definition we already mentioned the concept of *dependency*. We define two types of dependency a Boolean function can have:

- **Syntactic Dependency.** A Boolean function f is syntactically dependent on literal ℓ if $\ell \in \mathtt{VAR}(f)$.
- **Semantic Dependency.** A function f semantically depends on ℓ if there exist two assignments that differ only in ℓ and yield different evaluations of f. Semantic dependencies form a subset of syntactic ones.

We also define a desirable structural property for reset functions:

Stratified Reset. A set of reset functions R is *stratified* if they have no cyclic dependencies. To check this condition efficiently, we use the syntactic definition of dependency: construct the graph $G = (L, \{(u, v) \mid v \in \mathtt{VAR}(r_u)\})$ and check whether it is acyclic.

In AIGER circuits, this corresponds to ensuring that the graph—where each AND gate has edges to its inputs, and each latch has an edge to its reset—is acyclic.

The benefit of stratified resets is that a reset state always exists, since a model of $R\{L\}$ can be constructed by assigning latches in reverse topological order. More than that, given a subset K of the latches L, with an assignment s satisfying $R\{K\}$, if a topological ordering of L, ending in K exists, s can be extended to satisfy $R\{L\}$.

For transition functions, a similar argument comes more easily: Given $K \subseteq L$ and an assignment s to $L_0 \cup K_1$ that satisfies $F_{0,1}\{K\}$, s can be extended to an assignment over $L_0 \cup L_1$ satisfying $F_{0,1}\{L\}$.

These two restrictions—stratified reset and total transition functions—are what we claimed come naturally at the beginning of this section and allow us to make model checking certification more efficient. See Chapter 9 for further discussion.

2.4 Model Checking Certificates

The same arguments that make certificates necessary for SAT solving apply to model checking— arguably even more so, as model checkers are often composed of multiple engines running distributed across machines over the course of several days. Moreover, they frequently rely on composing multiple other formal tools to solve subproblems.

A significant part of this thesis is devoted to the question of what model checking certificates should look like, such that they cover a wide range of techniques and can be checked efficiently. For now, we only mention that the certificates themselves are also circuits, and checking their validity is reduced to checking the validity of five SAT formulas. Further, the construction of these formulas is linear in the size of both the model and the certificate. Details are presented in Chapter 9 and Chapter 10.

With these few details in place, let us consider the complexity implications of model checking certification: The class NP can be characterized as the set of problems for which a certificate exists that is polynomial in the input size and can be verified in polynomial time. SAT solving—that is, the question of whether a satisfying assignment exists—is the canonical NP-complete problem [52, 53], meaning it is as hard as any problem in NP. Conversely, proving that all assignments do *not* satisfy a formula (i.e., unsatisfiability or validity) is coNP-complete. Together with the class P, these classes form the first three levels of the *polynomial hierarchy*. Each additional level corresponds to adding one more quantifier to a SAT formula in an alternating fashion. At the top of this hierarchy lies PSPACE— the class of all problems solvable using polynomial space— whose canonical problem is quantified Boolean formula (QBF) evaluation [54].

We currently have no proof that this polynomial hierarchy is *real* at all, i.e., we do not know of a single problem that is in PSPACE but provably not in P. Nevertheless, showing that some result would make two levels of the hierarchy equally hard, and thus collapsing the hierarchy at that level is considered a strong indication that the result does not hold. For instance, we do not expect unsatisfiability certificates to be polynomial in size; if they were, NP would equal coNP, collapsing the polynomial hierarchy at the second level.

We can make a similar argument for model checking certificates: If a complete model checking algorithm produced polynomial-size witnesses for safe instances, the polynomial hierarchy collapses to the second level. To justify this result, we first show that model checking circuits is PSPACE-complete.

We can perform model checking with a nondeterministic Turing machine by constructing the initial state, nondeterministically choosing inputs, and computing the next state. Each transition increments a counter. Once this counter reaches $2^{|L|}$, the machine halts. If at any point a bad state is encountered, the machine accepts. By Savitch [55], the space requirements for a deterministic Turing machine only increase quadratically.

On the other hand, consider a closed QBF formula of the form

$$\forall x_n \exists x_{n-1} \ldots \exists x_0.\, f(x_n, \ldots, x_0),$$

where f is a Boolean function. We construct a circuit that is safe if and only if the formula is valid. Each variable x_i corresponds to a latch c_i, transitioning like a binary counter with c_0 as the least significant bit. Additionally, each x_i is associated with a bookkeeping latch b_i, initialized to $\top$ if x_i is universally quantified and to $\bot$ otherwise. Further:

$$f_{b_i} = \begin{cases} b_i, & \text{if } c_i \leftrightarrow f_{c_i} \\ \bot, & \text{if } c_i \wedge \neg f_{c_i} \\ q_i, & \text{otherwise} \end{cases} \qquad q_i = \begin{cases} b_i \vee q_{i-1}, & \text{if } x_i \text{ is existentially quantified} \\ b_i \wedge q_{i-1}, & \text{if } x_i \text{ is universally quantified} \end{cases}$$

where q_{-1} is identified with an encoding of $f(c_n, \ldots, c_0)$.

The bookkeeping latches b_i encode whether the formula is currently satisfied, considering only the quantifiers within the scope of x_i. The property $P = \vee_{i \in [0,n]} \neg c_i \vee b_n$ then fails if the full formula is not satisfied after the counter reaches saturation. Thus, exactly if the QBF formula is not valid. Since the QBF problem is PSPACE-complete [56], we have proven the same for our model checking problem.

Returning to the question of polynomial-size witnesses: Consider a nondeterministic Turing machine with access to an NP oracle. Such a machine could decide any model checking instance by guessing a polynomial-size certificate, constructing the associated SAT formulas, and querying the oracle to verify their validity.

We are therefore content with certificate constructions that are exponential in the worst case. Especially since their size tends to be very manageable in practice, as discussed in Chapter 11.

Finally, note that the circuit used to establish PSPACE-hardness does not use inputs. Thus, the construction also proves that ternary simulation (as defined in Chapter 8) is PSPACE-complete. It further implies that validating counterexample traces is also PSPACE-complete. The former motivates the *widening* technique introduced in Chapter 8, and the latter motivates future work on compact formats for counterexamples.

Chapter 3

Discussion on Published Work

This chapter provides an overview of the peer-reviewed papers presented in the remainder of the thesis and discusses their relation to each other. In addition to the two main topics that give the thesis its title—SAT and model checking— the evaluation of tools, particularly in the form of competitions, is a pervasive theme throughout. Figure 3.1 illustrates how each paper relates to these three central areas.

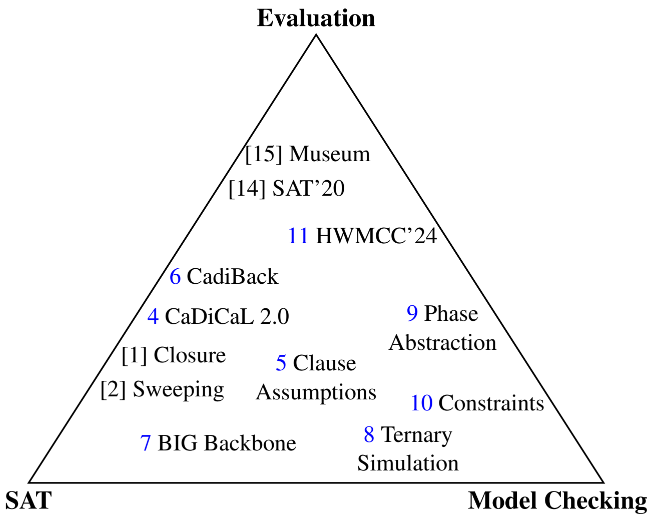

Figure 3.1: Overview of papers. Those with chapter marks are included in this thesis.

The foundation of this thesis lies in efficient SAT solvers. Because model checking inherently requires incremental SAT solving[1], the state-of-the-art incremental SAT solver CaDiCaL serves as the SAT backend in our model checker and other tools. Chapter 4 presents version 2.0 of CaDiCaL, focusing on proof production and incremental solving techniques. The paper includes a comprehensive evaluation of CaDiCaL and competing solvers across multiple incremental applications such as backbone extraction and bounded model checking.

There are two additional papers, of which I am a co-author but are not included in the thesis, which are highly relevant for the question of how to improve SAT solving for

[1]Since model checking is PSPACE-hard, we do not expected to solve it with a single SAT query in NP.

model checking: Clausal Congruence Closure [1] and Clausal Equivalence Sweeping [2]. The former extracts structural information from CNF and applies congruence closure to identify isomorphic subcircuits. The latter implements SAT sweeping [57] directly at the CNF level to detect equivalent gates. We include an evaluation of these techniques in Appendix 11.6, focusing on their impact on certificate validation.

The idea of optimizing SAT solving specifically for model checking is exemplified in Chapter 5, which presents an extension to the incremental SAT solver interface that speeds up a particular model checking algorithm. This extension was implemented in CaDiCaL and is now being considered for the next IPASIR interface revision [58]. While originally developed for a specific purpose, it has since gained widespread adoption.

Backbone extraction, while not directly associated with model checking, plays a useful role in some preprocessing techniques. Chapter 6 introduces the backbone extraction tool CADIBACK. It was originally developed because the third author needed to label backbones as a feature for a machine learning application. Since then it has become a critical component in every model counter participating in the competition, including the winner GANAK [59]. CadiBack reimplements existing algorithms using modern SAT technology, yielding significant performance improvements. Interestingly, clause assumptions—introduced for model checking—are heavily used in CadiBack and may be even more beneficial in this setting.

For our own work, it is often unnecessary to extract the complete backbone. Chapter 7 presents a polynomial algorithm for extracting the backbone from the binary implication graph, along with theoretical results that establish the completeness of the approach for a specific class of formula.

Ternary simulation is a core technique in model checking. Chapter 8 formalizes ternary simulation as abstract interpretation and introduces novel widening and narrowing operators based on clause subsumption and backbone information.

Phase abstraction is a preprocessing technique that unfolds a circuit along the temporal dimension and eliminates clock signals. It starts out by finding an overapproximation of the states reachable in certain clock cycles using ternary simulation. The generalized version presented in Chapter 9 makes use of the techniques introduced in Chapter 8 to consider multiple such approximations and select the one that minimizes circuit complexity. The main focus of this chapter is how to extend the trust we can put into a model checker using complex preprocessing techniques like phase abstraction and temporal decomposition.

Constraints restrict the set of admissible circuit states, which breaks some desirable properties we described at the end of Chapter 2. Chapter 10 proposes relaxed checks to allow complex constraints in certificates. However, this requires a shift from SAT to QBF-based checking, which is less scalable. Since most model checking algorithms can produce certificates without needing constraints, SAT-based checking remains the default. A “long-standing” problem in our very specific field of work was how to certify k-induction under uniqueness constraints [12], which Chapter 10 addresses. While the solution works, it requires quantified induction and does not scale well.

Our efforts on model checking certification culminated in the introduction of mandatory certificates to the hardware model checking competition in 2024. Despite our prior concerns, the competition sported more participants than ever before. The certificates produced were small, fast to check, and—most importantly—did not hinder performance: the winner outperformed the previous state of the art even when accounting for certificate validation. Details are presented in Chapter 11.

I contributed to two further works focused on competition and evaluation: a journal paper discussing the SAT competition in 2020 [14], and the SAT Museum [15], which compares current and historical SAT solvers. For these I provided a statistical evaluation of the similarity between solvers and further used statistical measures to judge the impact of the benchmark selection on the outcome of the competition.

The four papers mentioned here, but not included in this thesis, complement the overall presentation and provide additional context. They are omitted as the author's contribution to them was minor.

Chapter 4

CaDiCaL 2.0

Published In International Conference on Computer Aided Verification (CAV) 2024

Authors Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt

Changes from Published Version Corrected typos and adjusted layout.

Authors Contributions Most of the new features in CaDiCaL 2.0 presented after the introduction have been previously described in other papers, either by the development team or by other researchers. The author of this thesis implemented the constraint feature described in Chapter 5. He is also responsible for running, evaluating, and presenting the results of the experimental evaluation in this tool paper, including the adaptations to the new incremental applications introduced therein.

Acknowledgement This work was supported in part by the Austrian Science Fund (FWF) under project T-1306, W1255-N23, and S11408-N23, the state of Baden-Württemberg through bwHPC, the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG, the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract 16ME0132, and by a gift from Intel Corporation.

Abstract The SAT solver CaDiCaL provides a rich feature set with a clean library interface. It has been adopted by many users, is well documented and easy to extend due to its effective testing and debugging infrastructure. In this tool paper we give a high-level introduction into the solver architecture and then go briefly over implemented techniques. We describe basic features and novel advanced usage scenarios. Experiments confirm that CaDiCaL despite this flexibility has state-of-the-art performance both in a stand-alone as well as incremental setting.

4.1 Introduction

Progress in SAT solving has a large impact on model checking, SMT, theorem proving, software- and hardware-verification, and automated reasoning in general, and, according to "The SAT Museum" [15], SAT solvers get faster and faster, at least on benchmarks consisting of a single formula. For incremental SAT solving it was less clear, particularly as preprocessing [60] and inprocessing [61] heavily contributing to this improvement were considered incompatible with incremental solving (the winners of the SAT competition main track rely on inprocessing since 2009 except in 2011/2012/2016 and since 2005 all on preprocessing).

A simple and elegant solution to this problem is due to the award winning incremental SAT solving approach [62] first implemented in CADICAL. It reverts clause removal, i.e., restores clauses removed during pre- and inprocessing, restrictively on a case-by-case basis. It allows incremental solving to make full use of pre- and inprocessing techniques, in contrast to less general solutions [63–66], without reducing their effectiveness nor burden the user to "freeze" and "melt" variables ("*Don't Touch*" variables in [67]) as necessary with MINISAT [68].

This is the first tool paper on CADICAL, while previous, actually well cited, descriptions appeared only as system description in non-peer-reviewed SAT competition proceedings [69–74]. In general, even though "SAT is considered a killer app for the 21st century" (Donald Knuth), there are few tool papers on SAT solvers, with the prominent exception of MINISAT [68], which appeared in 2003 and was awarded the test-of-time award at SAT'22. The descriptions of CRYPTOMINISAT [75], GLUCOSE [76] and INTELSAT [77] introduce the corresponding SAT solver and can be considered to be tool papers too though.

Development of CADICAL was triggered by discussions at the "Theoretical Foundations of SAT Solving Workshop" in 2016 at the Fields Institute in Toronto, where it became apparent that both theoreticians and practitioners in SAT have a hard time understanding how practical SAT solving evolved, what key components there are in modern SAT solvers and, most importantly, that it was apparently getting harder and harder to modify state-of-the-art solvers for controlled experiments or to try out new ideas. With CADICAL we tried to change this, thus the main objective was to produce a clean solver, with well-documented source code, which is easy to read, understand, modify, test, and debug, without sacrificing performance too much.

The first goals were achieved from the beginning and performance improved over the years. After its introduction in 2017 CADICAL continued to achieve high rankings in yearly SAT competitions, e.g., in 2019 it solved the largest number of instances in the main track, but scored less than the winner. It never won though except for the most recent SAT competition in 2023 where CADICAL was combined with a strong preprocessor employing bounded variable addition [41, 78]. The competition organizers paraphrased this as "CADICAL strikes back".

Moreover, with the show-case of our new incremental approach [62] we invested in increasing the feature set supported by CADICAL culminating for now in supporting "user propagators". This for instance allowed to replace the original but highly modified

MiniSat based SAT engine in cvc5 by CaDiCaL, as described in a recent well-received SAT'23 paper [79].

The users of CaDiCaL fall into three categories. A first group applies the solver out of the box on benchmarks where CaDiCaL turns out to have superior performance. As an example consider solving mathematical problems with the help of SAT solving such as [80–83]. Second, there is an increasing user base, including [6, 62, 79, 84–89], which relies on the rich application programmable interface (API) provided by CaDiCaL, particularly its incremental features. Third, there are research prototypes modifying or extending CaDiCaL to achieve new features, including [41, 90–94]. Some of these modifications have been integrated [43, 95] but others remain future work [41].

Finally, CaDiCaL is used as a blue-print for understanding, porting, and integrating state-of-the-art techniques into other solvers. In this regard we are in contact with companies in cloud services, hardware design, and electronic design automation. It was also consulted in developing IsaSAT [96], the only competitive fully verified SAT solver. Furthermore CaDiCaL was adopted as template solver for the "hack track" of the yearly SAT competition since 2021 as an "easy to hack" state-of-the-art SAT solver.

Related SAT solvers in the SAT competition often lack documentation, are hard to extend and modify, and, most importantly, do not provide such a rich and clean library interface as CaDiCaL. For instance our SAT solver Kissat [72] falls into this category. It has been dominating the SAT competition 2020–2022 (in 2022 all top-ten solvers were descendants of Kissat), is more compact in memory usage and often faster on individual instances, but is lacking support for even the most basic incremental features such as assumptions.

The majority of the solvers in the SAT competition are restricted in their feature set as they are tuned for stand-alone usage, i.e., running the solver on a single formula stored in a file in DIMACS format [97], even though there is occasionally an incremental track in the SAT competition (last one that really took place was in 2020 as the one announced in 2021 was later cancelled).

Prominent SAT solvers with a richer feature set and particularly supporting incremental solving, beside the rather out-dated MiniSat [68], are newer versions of CryptoMiniSat [75], and Glucose [98]. The former is actively developed and in terms of implemented techniques has quite some overlap with CaDiCaL. In addition it offers special support for XOR reasoning, solution sampling and model counting [99]. The Glucose solver has been improved for incremental solving [100] but is not comparable in terms of implemented techniques nor features.

Unique and non-common features of CaDiCaL include: literal flipping [6], single clause assumption [5], incremental solving without freezing [62], extensive logging support, record & play of API calls, model-based testing, internal proof and solution (model) checking, termination and clause learner interfaces, various preprocessing techniques, an online proof tracing interface, formula extraction (after simplification), support of many external proof formats (DRAT, LRAT, FRAT, VeriPB) [43], and last but not least the user propagator [79].

This paper is structured by describing in the next section the architecture of CaDiCaL, which also acts as a summary of integrated techniques and provided features. The rest

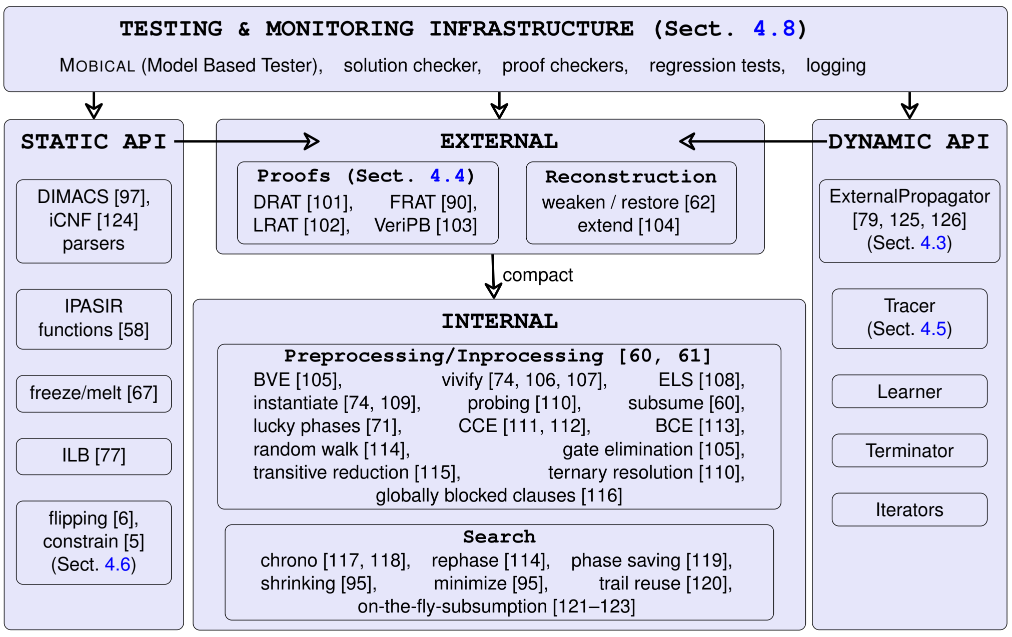
TESTING & MONITORING INFRASTRUCTURE (Sect. 4.8)
MOBICAL (Model Based Tester), solution checker, proof checkers, regression tests, logging
STATIC API
DIMACS [97], iCNF [124] parsers
IPASIR functions [58]
freeze/melt [67]
ILB [77]
flipping [6], constrain [5] (Sect. 4.6)
EXTERNAL
Proofs (Sect. 4.4)
DRAT [101], FRAT [90], LRAT [102], VeriPB [103]
Reconstruction
weaken / restore [62] extend [104]
compact
INTERNAL
Preprocessing/Inprocessing [60, 61]
BVE [105], vivify [74, 106, 107], ELS [108], instantiate [74, 109], probing [110], subsume [60], lucky phases [71], CCE [111, 112], BCE [113], random walk [114], gate elimination [105], transitive reduction [115], ternary resolution [110], globally blocked clauses [116]
Search
chrono [117, 118], rephase [114], phase saving [119], shrinking [95], minimize [95], trail reuse [120], on-the-fly-subsumption [121–123]
DYNAMIC API
ExternalPropagator [79, 125, 126] (Sect. 4.3)
Tracer (Sect. 4.5)
Learner
Terminator
Iterators

Figure 4.1: An overview of the main components of CADICAL.

of the paper consists of highlighting recently added features of the solver or features not presented before, followed by experiments showing that CADICAL has state-of-the-art performance, before concluding.

4.2 Architecture

CADICAL is a modern SAT solver with many features written in C++. It can be used as stand-alone application through the *command-line interface* (CLI) or as library through its *application programming interface* (API) in C++ (or in limited form in C). Fig. 4.1 depicts a structural overview. The central component, called `Internal`, implements CDCL search [127, 128] and formula simplification techniques [60, 61]. On top of it, the `External` facade hides the internals while maintaining the proofs and solutions (aka *models*) of solved problems.

The heart of the solver is the function `cdcl_loop_with_inprocessing` in `Internal` which interleaves the CDCL loop with formula simplification steps (i.e., with inprocessing [61]). During `Search`, CADICAL supports several techniques, like chronological backtracking [117, 118], rephasing [114], and shrinking [95], which are only some of the important features. See Fig. 4.1 for more references.

The CDCL loop [128] is scheduled to be preempted in regular intervals to let the solver apply various formula simplification [60] and inprocessing techniques [61]. Each technique is implemented separately (e.g., in file `subsume.cpp`) and has (i) a corresponding function which determines if the solver should preempt CDCL search and apply the technique (e.g., `subsuming()`) and (*ii*) a function that actually applies the technique (e.g., `subsume()`).

As Fig. 4.1 shows, CADICAL implements a variety of preprocessing/inprocessing techniques, including bounded variable elimination (BVE) [105], arguably the most effective one. As further examples, CADICAL also supports vivification [106, 107] and instantiation [109]. Combining them [74] won the CADICAL "hack track" 2023.

The `External` component communicates with `Internal` by mapping active variables into a consecutive sequence of integers (*compacting*) and extends internal solutions back to complete solution of the input problem with the help of the reconstruction stack [104]. In incremental use cases `External` also keeps the reconstruction stack *clean* [62] by "undoing" previous inprocessing steps. Beyond that, `External` connects internal and external proof generation (see Sect. 4.4).

We distinguish two types of API usage in CADICAL: *static* and *dynamic*. The static API provides access to standard solver functionalities *between* SAT solving calls (like IPASIR [58], parsing DIMACS, or iCNF files). With ILB as proposed by INTELSAT [77], we try to keep the trail unchanged between incremental calls.

The dynamic API interacts and controls the solver *during* `Search`. The solver provides dynamic access to clauses learned during conflict analysis to connected `Learner` instances. The `Terminator` class interface allows users to asynchronously terminate the solving procedure. Through the `Iterator` interface of CADICAL, the user can iterate over the irredundant (simplified) clauses of the problem or can iterate through clauses on the reconstruction stack, supporting simplified formula extraction and external model reconstruction.

4.3 External Propagator

Applications of CADICAL, for example within the SMT solver cvc5 [87] (and maybe in the future within other lazy SMT solvers, such as Z3 [129] or Yices [130]), or to support Satisfiability Modulo Symmetries (SMS) [79, 131], require more control over the solver than provided by the standard incremental IPASIR interface [58]. To this purpose CADICAL supports a more fine-grained and tighter integration into larger systems by allowing an external user propagator [125, 126, 132] to be connected to it through the IPASIR-UP interface [79].

This abstract interface is defined in the `ExternalPropagator` class which provides corresponding notification and callback functions. Inheriting from this class allows users to implement dedicated external propagators which for instance import and export learned clauses or suggest decisions to the SAT solver. The full description of functionalities supported by the IPASIR-UP interface is available in [79]. Here we focus on CADICAL-specific implementation details.

First, CADICAL ensures that only external variables appear in the IPASIR-UP interactions, thereby allowing users to ignore the internal (compacted) details. Furthermore, CADICAL employs preprocessing and inprocessing even when an external propagator is connected. To avoid the need to restore clauses during the CDCL loop and to ensure that solution reconstruction [104] does not change assignments of *observed variables* (i.e., relevant to the external propagator), every observed variable is automatically frozen. As a side effect, the external propagator can only set *clean* [62] variables as new observed variables during search. As fresh variables are always clean, this is acceptable and mostly sufficient in practice.

Finally, CADICAL, by default, considers every external clause as irredundant, exactly as the original input clauses of the problem. Thus, during clause database reduction they are not candidates for removal and so can be deleted only when implied by the rest of the formula. In future work we plan to allow users to specify the redundancy of the external clauses and to support incremental inprocessing [62] even for variables observed by the external propagator.

4.4 Proofs

Unsatisfiability proof certificates are an integral part of SAT solving [133, 134]. Even though clausal proofs were introduced in 2003 [68, 135], checking large proofs only became viable with deletion information [40]. The most prominent format today is DRAT [101] which was mandatory in the SAT competition from 2016 [136] to 2022. In 2023 both DRAT [101] and VeriPB [103] were allowed in the competition [137].

The proof formats GRAT [138] and LRAT [102] were proposed to allow even faster proof checking, i.e., by trading time for space, but also to facilitate formally verified proof checkers (e.g., CAKE_LPR [39]). They require hints for clause additions in form of antecedent *clause identifiers* (ids). External tools like DRAT-TRIM [101] can add such hints in a post-processing step to DRAT proofs.

The proof formats DRAT [101], FRAT [90], LRAT [102], and VeriPB [103] are supported by CADICAL. It is the first solver to support LRAT natively. Without the need for post-processing this reduces proof checking time [43] substantially.

Recent diversification of proof formats in the SAT competition [137] motivated us to add VeriPB. It is a general proof format for various applications [103, 139, 140]. The tool-chain for checking SAT solver proofs with the verified VeriPB backend [103] is under development and not fast enough yet. Actually, BREAKID-KISSAT [137], one of the top performers in the SAT competition 2023, lost due to multiple timeouts during proof checking. Similarly to FRAT, CADICAL can provide antecedents in VeriPB proofs. We expect this to speed up VeriPB proof checking considerably.

4.5 Tracer Interface

The dynamic API allows to extract proof information from CADICAL online without files by connecting user-defined tracers as instances of the virtual C++ class `Tracer`. It provides notifications and callbacks for proof-related events, such as addition and deletion of clauses. Proof writers for all formats (Sect. 4.4) as well as both internal proof checkers (Sect. 4.8) go through the `Tracer` class. Furthermore, there is ongoing

```
void add_derived_clause (uint64_t new_id, bool redundant,
                         const vector<int> & literals_of_clause,
                         const vector<uint64_t> & antecedent_ids);
```

Figure 4.2: `Tracer` virtual callback function to add a derived clause to the proof.

work to produce VeriPB proofs for the MaxSAT solver Pacose [141] using the `Tracer` interface in CADICAL.

We support a large set of event types covering a multitude of use cases. Information provided includes antecedent ids and literals of clauses, separation between original, derived, and restored [62] clauses, and information of clause redundancy, as well as weakening [61] and strengthening [61, 103]. For example, Fig. 4.2 shows the callback function for the proof event of adding a derived clause, where "`derived`" means entailed by the formula (i.e., not original input clause). Additional notifications include reserving ids for original clauses, as used for generating file based proof formats, such as VeriPB and LRAT.

For each solve call, a concluding event gives precise information about the result: a model concludes satisfiable instances, whereas for unsatisfiable instances we provide information about the final conflict clause. We have recently started to explore incremental proof tracing as well [142, 143].

4.6 Constraints and Flipping

SAT solvers are used in a wide range of applications in many different ways. For incremental solving, MINISAT has been the predominant choice. However, in recent years, CADICAL has begun to replace MINISAT in numerous applications, most prominently cvc5. This can be attributed to its overall better performance and various application-specific features unique to CADICAL.

A prime example is the *constraint* feature [5], which allows users to define a temporary clause with the same lifespan as assumptions. It was initially developed to support the SAT based model checking algorithm IC3 [49], which requires often millions of incremental SAT calls during a single run, where each query needs to assume a single clause valid only for that call.

Constraints do not introduce new functionality per se, as temporary clauses can be simulated by activation literals. But they do allow the solver to employ a more efficient implementation, as they particularly avoid to introduce those assumption variables. Beyond IC3, constraints have also proven useful in our backbone extractor CADIBACK [6]. The purpose of using *constraint* in backbone extraction is to find maximally diverging models in order to eliminate backbone candidates fast. CADIBACK uses constraints to ensure that each new model includes at least one literal not observed in previous models. If this is not possible, all unseen literals are immediately determined to be in the backbone.

Once a model is found, we use another feature called *literal flipping* [144] to eliminate further backbone candidates [6]. A literal is *flippable* if toggling its value also results in a model. This concept was employed to speed-up backbone MiniBones [145] before and also MUS extraction [146]. In these earlier works it was implemented by iterating over all clauses outside the SAT solver, searching for literals that can be flipped in the model provided by the solver. Using clause watching our implementation inside of CaDiCaL is much more efficient.

4.7 Interpolation

Software-based test generation targeting RISC-V in the Scale4Edge project [147] relied on interpolation-based model checking and MiniCraig to generate interpolants. It uses MiniSat as SAT solver and in this application constitutes a performance bottleneck. Therefore we developed a new more scalable solver CaDiCraig based on CaDiCaL and its proof tracer API (Sect. 4.5).

The implementation of CaDiCraig is external to CaDiCaL. It uses the same interpolant construction as in MiniCraig but is now separated from MiniSat. We are not aware of any other modern open-source SAT solver which allows to build interpolants through a generic API without being forced to write the whole proof to a file, trimming and post-processing it on disk, such as in [148].

The CaDiCraig tracer constructs partial interpolants as usual, e.g., see [149]. Through the proof tracer API the tracer is notified by CaDiCaL about each new clause and its antecedents needed to derive it by resolution. It then builds a partial interpolant for that clause using previously computed partial antecedent interpolants. When the solver concludes deriving an empty clause and thus showing unsatisfiability (Sect. 4.5) the final interpolant is built from the antecedents of the empty clause. It can then be retrieved by via the CaDiCraig API.

4.8 Testing and Debugging

Such a sophisticated and complex software as CaDiCaL necessitates rigorous testing to ensure correctness of interactions between its multitude of features. In this section we discuss our arsenal of essential testing and debugging techniques.

First, we primarily rely on logging for debugging purposes. For instance, when enabled, CaDiCaL will print every single step from its creation to its deletion. From an implementation perspective, logging features are not compiled in by default to avoid performance overhead in release builds. Furthermore, if enabled at run-time, CaDiCaL prints verbose information about the inprocessing schedule, useful for debugging performance regressions (e.g., inprocessor scheduling).

Further useful debugging tools are the built-in checkers. The LRAT and DRAT checkers are optional and ensure that every learned clause is properly derived. The new LRAT checker [43] was crucial for achieving LRAT support.

Last but not least we want to mention the API fuzzer Mobical, which generates random API calls and minimizes failing runs. Internally, Mobical implements a state machine issuing API calls. It also performs option fuzzing by varying available options. This approach is extremely useful to produce short failing API call traces focusing on the actual defect, e.g., like picking a low garbage collection limit to trigger a defect in the garbage collector. Combining checkers with MOBICAL greatly increases its strength. During development it is advisable to build MOBICAL and CADICAL with assertions and checkers enabled.

MOBICAL is similar in spirit to the related model-based tester of LINGELING [150] for SAT and BTORMBT [151] and MURXLA [152] targeting SMT. Note that other SMT fuzzers [153–156] focus on non-incremental usage or only support incremental "push & pop" [157]. For non-incremental SAT solving, there is also CNFFUZZ fuzzer and the CNFDD delta-debugger [150, 158].

Accordingly, we have implemented a MOCKPROPAGATOR class in MOBICAL to test the EXTERNALPROPAGATOR API. It fuzzes the IPASIR-UP implementation in combination with all options and features of the solver. It revealed several corner-cases which we believe would have been very hard to trigger otherwise.

MOBICAL targets only incremental SAT problems and could not help when incorrect interpolants showed up in earlier experiments with MINICRAIG and CADICRAIG. Therefore, we have built an external interpolation fuzzer in Python. It checks interpolants and an accompanying delta-debugger minimizes problems by deleting command line options, clauses, and variables.

4.9 Experiments

The performance of CADICAL 2.0 was evaluated in three experiments. We first follow the *non-incremental* setup of the main track of the SAT competition, where solvers are run on benchmark files in DIMACS format. The second experiment focuses on *incremental* usage, i.e., following the incremental track of the competition. Finally we show the effectiveness of CADICAL in the context of *interpolation* via its `Tracer` API. All experiments were conducted on our cluster with Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled).

Non-Incremental. The winner SBVA-CADICAL [159] of the main track of the SAT Competition 2023 combined a novel technique for bounded variable addition [78] with CADICAL 1.5.3. In their implementation preprocessing was limited to 200 seconds which yields different preprocessed formulas over multiple runs. Therefore, we ran the preprocessor of SBVA-CADICAL separately for 200 s, and then gave the same formulas to CADICAL 1.5.3 and our new version CADICAL 2.0. Running them for 5 000 s as in the competition (ignoring preprocessing time in essence) gave very similar results. We provide more details in the artifact and in the appendix. This confirms that CADICAL (also in version 2.0) is state-of-the-art in non-incremental solving.

Incremental. How to assess the incremental performance of a SAT solver is less established. To present an unbiased evaluation, we follow the principles set out by the last incremental track of the SAT competition in 2020 [14]: The solvers are evaluated in six different applications, each featuring 50 benchmarks, with a 2 000 s timeout and 24 GB memory limit. Four applications are carried over directly from the 2020 competition: the CEGAR-based QBF solver IJTIHAD, the simple backbone extractor BONES, the longest simple-path search LSP, and the MaxSAT solver MAX. However, we exclude two applications: the essential variable extractor and the classical planner PASAR. Both use features that are not present in all solvers. The former queries `ipasir_learned`, which is missing from CADICAL 1.0, and the latter relies on limiting the number of conflicts. Instead, we include the bounded model checker for bit-level hardware designs CAMICAL [62] and the sophisticated backbone extractor CADIBACK [6].

The benchmarks from the incremental track from the 2020 SAT Competition remain unchanged. For CAMICAL, we randomly select 50 Boolean circuits used in HWMCC'17 [160]. Although CADIBACK solves the same problem as BONES, we opt for a distinct set of benchmarks. In 2020 the "smallest and easiest satisfiable" [14] CNF formulas were selected and even though backbone extraction is harder than mere solving, they were rather easy. Conversely, we compile a non-trivial set of benchmarks by randomly selecting satisfiable formulas from past competitions (2004-2022)[6] that take KISSAT 3.0.0 [144] more than 20 s to solve. We use KISSAT as it is not incremental and hence does not compete.

The artifact has a comparison of CRYPTOMINISAT and CADICAL on 1 798 formulas [6] and indicates that our selection does not impact the outcome. See Fig. 4.3 in the appendix for further discussion. As detailed in Sect. 4.6, CADIBACK utilizes constraints, which are only available in recent versions of CADICAL and are simulated with activation literals otherwise.

Our evaluation includes all solvers that competed in 2020: ABCDSAT i20 [161] CRYPTOMINISAT 5 (CMS) [162, 163], and RISS 7.1.2 [164] . The CADICAL version from that year is referred to as CADICAL 2020. The other two versions are 1.0 from 2019 and our latest release 2.0. We also include MINISAT 2.2 and the latest version of GLUCOSE 4.2.1. Table 4.1 presents for each SAT solver and application: the PAR2 score, which is the average runtime in seconds with a penalty of 4000 for unsolved instances; and the number of solved instances.

Our results show that CADICAL 2.0 reaches state-of-the-art performance, demonstrating a distinct improvement over previous versions. Also, differing from the findings in [165], we see a significant advantage of the newer CADICAL and CRYPTOMINISAT, over the older MINISAT, further substantiated below.

Interpolants. To validate CADICRAIG using CADICAL, we converted all 400 benchmarks of the SAT Competition 2023 into interpolation problems split into A and B parts chosen with the goal to assign related clauses to the same part in order to keep the number of global variables limited. The index of the smallest variable of each clause determines the probability of the clause being assigned to A. On our crafted benchmarks (5 000 s timeout, 7 GB), CADICRAIG significantly outperforms MINICRAIG, solving 117 benchmarks, compared to only 75.

		CaDiBack	CaMiCaL	Bones	LSP	Max	Ijtihad	Total
CaDiCaL	2.0	$\mathbf{3297}_{\mid\underline{11}}$	$\mathbf{2606}_{\mid\underline{18}}$	$494_{\mid\underline{45}}$	$1898_{\mid\underline{27}}$	$\mathbf{1976}_{\mid\underline{26}}$	$\mathbf{2980}_{\mid\underline{13}}$	$\mathbf{2209}_{\mid\underline{140}}$
	2020	$3409_{\mid\underline{9}}$	$2677_{\mid\underline{17}}$	$622_{\mid\underline{43}}$	$1955_{\mid\underline{26}}$	$2015_{\mid\underline{25}}$	$2986_{\mid\underline{13}}$	$2277_{\mid\underline{133}}$
	1.0	$3495_{\mid\underline{7}}$	$2627_{\mid\underline{18}}$	$595_{\mid\underline{44}}$	$2011_{\mid\underline{26}}$	$2028_{\mid\underline{25}}$	$2989_{\mid\underline{13}}$	$2291_{\mid\underline{133}}$
	CMS	$3491_{\mid\underline{8}}$	$2701_{\mid\underline{17}}$	$\mathbf{397}_{\mid\underline{46}}$	$\mathbf{1773}_{\mid\underline{29}}$	$2021_{\mid\underline{25}}$	$3057_{\mid\underline{12}}$	$2240_{\mid\underline{137}}$
	MiniSat	$3678_{\mid\underline{5}}$	$2807_{\mid\underline{16}}$	$687_{\mid\underline{43}}$	$1993_{\mid\underline{26}}$	$2094_{\mid\underline{24}}$	$3123_{\mid\underline{11}}$	$2397_{\mid\underline{125}}$
	Riss	$3665_{\mid\underline{6}}$	$2836_{\mid\underline{15}}$	$892_{\mid\underline{40}}$	$1835_{\mid\underline{28}}$	$2017_{\mid\underline{25}}$	$3140_{\mid\underline{11}}$	$2398_{\mid\underline{125}}$
	abcdSat	$3582_{\mid\underline{7}}$	$2966_{\mid\underline{13}}$	$535_{\mid\underline{46}}$	$2493_{\mid\underline{21}}$	$2037_{\mid\underline{26}}$	$3207_{\mid\underline{10}}$	$2470_{\mid\underline{123}}$
	Glucose	$3778_{\mid\underline{4}}$	$2981_{\mid\underline{13}}$	$948_{\mid\underline{40}}$	$2078_{\mid\underline{25}}$	$2117_{\mid\underline{24}}$	$3206_{\mid\underline{10}}$	$2518_{\mid\underline{116}}$
	VBS	$3127_{\mid\underline{14}}$	$2546_{\mid\underline{19}}$	$257_{\mid\underline{48}}$	$1765_{\mid\underline{29}}$	$1856_{\mid\underline{28}}$	$2896_{\mid\underline{14}}$	$2075_{\mid\underline{152}}$

Table 4.1: Performance comparison of six incremental solvers, with three versions of CaDiCaL (2 000 s timeout). For each solver, we report PAR2 score over 50 benchmarks per application and number of solved instances ("PAR2$_{\mid\underline{\#\text{solved}}}$"). The four applications to the right have been used in the incremental track of the 2020 SAT competition. The best results per application are marked in **bold**. The last row presents the hypothetical *Virtual Best Solver* which always picks the best performing backend for each instance.

4.10 Conclusion

In this very first conference paper on CaDiCaL we reviewed its most important components and features as well as its testing and debugging infrastructure. We highlighted its use as SAT engine in SMT solving via the user propagator interface and how the tracer API can be used to compute interpolants. Our experiments show that CaDiCaL remains efficient despite this flexibility.

Producing incremental proofs is ongoing work [142, 143]. Further future work consists of producing incremental proofs for all features supported by CaDiCaL, avoiding to freeze observed variables by the user propagator, and porting into the main branch features provided by other users.

4.11 Appendix

4.11.1 Availability and Release Plan

We used version CaDiCaL 2.0.0-rc.5 (tagged with `rel-2.0.0-rc.5`) in our experiments. It is available on the development branch of CaDiCaL on GitHub at https://github.com/arminbiere/cadical with commit hash `a3ebcea`).

For the artifact and final release 2.0 we will further improve our latest additions for incremental proofs [142] and particularly also work on performance optimization for the SAT competition 2024.

4.11.2 Incremental Lazy Backtracking (ILB)

ILB and *reimply* are features proposed by INTELSAT [77]. The main idea of ILB is to keep as many assignments as possible on the trail between incremental solve calls while also working with assumptions. Nadel [77] claims that his novel reimply procedure increases the potency of ILB.

We implemented both ILB and reimply in CADICAL. However, in our experiments we see little variance between enabling and disabling any of the two features (Table 4.2). As reimply adds increased code complexity substantially we removed it again for version 2.0. We are still convinced, that ILB can be helpful for MaxSAT and similar applications and thus keep it.

	ILB	Reimply	CaMiCaL	CaDiBack	Bones	LSP	Max	Ijtihad	Total
2.0	✓	×	$2548_{\|19}$	$3318_{\|10}$	$477_{\|45}$	$1897_{\|27}$	$1970_{\|26}$	$2979_{\|13}$	$2198_{\|140}$
	×	×	$2548_{\|19}$	$3328_{\|10}$	$476_{\|45}$	$1898_{\|27}$	$1970_{\|26}$	$2986_{\|13}$	$2201_{\|140}$
1.9	✓	×	$2542_{\|19}$	$3268_{\|11}$	$466_{\|45}$	$1898_{\|27}$	$1975_{\|26}$	$2981_{\|13}$	$2189_{\|141}$
	×	×	$2543_{\|19}$	$3270_{\|11}$	$467_{\|45}$	$1898_{\|27}$	$1977_{\|26}$	$2988_{\|13}$	$2190_{\|141}$
	✓	✓	$2558_{\|19}$	$3335_{\|10}$	$493_{\|45}$	$1897_{\|27}$	$1976_{\|26}$	$2995_{\|13}$	$2209_{\|140}$
	×	✓	$2560_{\|19}$	$3379_{\|9}$	$492_{\|45}$	$1896_{\|27}$	$1980_{\|26}$	$2999_{\|13}$	$2218_{\|139}$

Table 4.2: Investigation of Incremental Lazy Backtracking (ILB) and reimply features of CADICAL. Reimply has been removed from the code base in version 2.0. However, ILB is enabled by default in version 2.0 and therefore matches the entry in Table 4.1.

4.11.3 Details on the CADIBACK Evaluation

In the original CADIBACK paper [6] it was evaluated on a large number of benchmarks, actually 1798 satisfiable SAT formulas collected from the SAT competitions between 2004 and 2020. We evaluated the two best solvers on the same set. As can been seen in Fig. 4.3 CADICAL outperforms CRYPTOMINISAT on this large set of benchmarks, suggesting that the selected subset presented in Table 4.1 is representative.

4.11.4 Testing and Debugging

To activate logging, the program needs to be configured with the command `./configure -l` before being compiled (with `make`). Note that logging still needs to be enabled at run time (with `-l`) and can then produce a substantial amount of output. An excerpt of debugging output for the "Tie & Shirt" example in the next section is shown in Fig. 4.7.

To produce verbose inprocessor scheduling information, CADICAL must be executed with `-v -v -v`. Here is an example of verbose output:

```
c [stabilizing-1] reached stabilization limit 3001 after 3001 conflicts
c [stabilizing-1] new stabilization limit 7001 at conflicts interval 4000
c [collect-3] moving 93088600 bytes of 2635278 non garbage clauses
c [collect-3] collected 96 bytes 0% of 2 garbage clauses
```

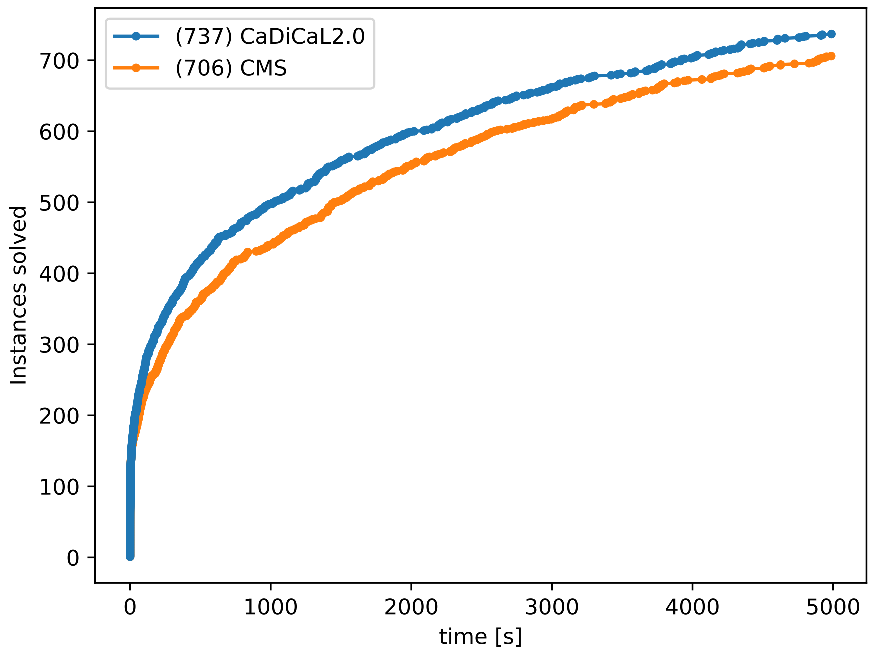

Figure 4.3: Number of benchmarks solved over time by CADIBACK with CADICAL 2.0 and CRYPTOMINISAT as backend SAT solvers. The benchmark set contains 1798 formulas.

```
c C  8.78 650 961 2 125 3001 2878 39% 2 2632400 819492 70%
c [compact-1] reducing internal variables from 1162410 to 819493
c [rephase-2] reached rephase limit 3001 after 3002 conflicts
```

In Fig. 4.4 we provide and example of running MOBICAL. It runs and minimizes each API call trace. A minimized trace can be executed with MOBICAL from within a symbolic debugger such as `gdb` to obtain more information about the failure. In this example we implanted an explicit `COVER(true)` coverage goal in the code, which work in the same way as assertions, and even in production code (with all other assertions disabled) triggers an `SIGABRT` signal when the condition is triggered. This is useful during development to produce call traces that reach a certain part of the code or making sure that a specific condition is not reachable which might help to reduce and simplify code size.

4.11.5 Tie & Shirt Demo

This example has been used by the first author in various talks as introduction on how to use SAT solvers. The following DIMACS file has three clauses, each encoding a constrain in the context of satisfying some dress code rules. The first clause encodes

```
# ./build/mobical
[...]
m count    seed/buggy/reducing/reduced      calls vars clauses
m -----------------------------------------------------------
m  5      bug-10705263437599747538.trace     1247   54 190    3
m  5      red-10705263437599747538.trace       44   11 11     1
^C
$ gdb --args ./build/mobical red-10705263437599747538.trace
[...]
cadical: shrink_block:319: ../src/shrink.cpp: Coverage goal 'true' reached.
```

Figure 4.4: Here MOBICAL was launched to check if some condition can be triggered (by adding a `COVER(true)`). It found one bug with seed 10705263437599747538 and minimized it to the file `red-10705263437599747538.trace` that we replay in gdb.

that wearing a tie (encoded as integer `1`) requires to wear a shirt too (encoded as integer `2`). The second one makes sure that least a tie or shirt is worn and finally the last clause says, that wearing a tie and a shirt is overkill.

```
p cnf 2 3
-1  2 0
 1  2 0
-1 -2 0
```

These rules are satisfiable and we can use CADICAL to determine a satisfying assignment which is shown in Fig. 4.6. The produced model shows that wearing a shirt but no tie satisfies the constraint. The same problem can further be solved through the API as explained in Fig. 4.5, actually in a more elaborated way.

4.11.6 CDCL Loop and Inprocessing

It was mentioned in Sect. 4.2 that the whole CDCL search and inprocessing in CADICAL is controlled by the `cdcl_loop_with_inprocessing` function. Fig. 4.8 shows most and actually literally the steps of this function (the logging and debugging specific parts are omitted).

When `solve()` is called, CADICAL first applies preprocessing (including local search and lucky phases), and, in case it does not find any solution, the function `cdcl_loop_with_inprocessing` is called. The function sums up what is checked and in which order during search and inprocessing in CADICAL. Note that here vivification is called as part of forward subsumption (`subsume()`).

```
CaDiCaL::Solver * solver = new CaDiCaL::Solver;

// Encode Problem and check without assumptions.

enum { TIE = 1, SHIRT = 2 };

solver->add (-TIE), solver->add (SHIRT),  solver->add (0);
solver->add (TIE),  solver->add (SHIRT),  solver->add (0);
solver->add (-TIE), solver->add (-SHIRT), solver->add (0);

int res = solver->solve ();  // Solve instance.
assert (res == 10);          // Check it is 'SATISFIABLE'.

res = solver->val (TIE);     // Obtain assignment of 'TIE'.
assert (res < 0);            // Check 'TIE' is 'false'.

res = solver->val (SHIRT);   // Obtain assignment of 'SHIRT'.
assert (res > 0);            // Check 'SHIRT' is 'true'.

// Incrementally solve again under one assumption.

solver->assume (TIE);        // Now force 'TIE' to true.

res = solver->solve ();      // Solve again incrementally.
assert (res == 20);          // Check it is 'UNSATISFIABLE'.

res = solver->failed (TIE);  // Check 'TIE' responsible.
assert (res);                // Yes, 'TIE' in core.

res = solver->failed (SHIRT);// Check 'SHIRT' responsible.
assert (!res);               // No, 'SHIRT' not in core.

// Incrementally solve once more under another assumption.

solver->assume (-SHIRT);     // Now force 'SHIRT' to false.

res = solver->solve ();      // Solve again incrementally.
assert (res == 20);          // Check it is 'UNSATISFIABLE'.

res = solver->failed (TIE);  // Check 'TIE' responsible.
assert (!res);               // No, 'TIE' not in core.

res = solver->failed(-SHIRT);// Check '!SHIRT' responsible.
assert (res);                // Yes, '!SHIRT' in core.

delete solver;
```

Figure 4.5: Tie & Shirt example of API usage (from "`cadical/test/api/example.cpp`").

```
c --- [ banner ] -------------------------------------------------------
c
c CaDiCaL SAT Solver
c Copyright (c) 2016-2023 A. Biere, M. Fleury, N. Froleyks, K. Fazekas, F. Pollitt
c
c Version 2.0.0-rc.5 a3ebcea9903f7b71d28fb5133d81d5592b05089a
c g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0 -Wall -Wextra -O3 -DNDEBUG
c Fri Jan 26 11:02:24 CET 2024 Linux ktp14s 6.5.0-14-generic x86_64
c
c --- [ parsing input ] ------------------------------------------------
c
c reading DIMACS file from 'tie.cnf'
c opening file to read 'tie.cnf'
c found 'p cnf 2 3' header
c parsed 3 clauses in 0.00 seconds process time
c
c --- [ options ] ------------------------------------------------------
c
c all options are set to their default value
c
c --- [ solving ] ------------------------------------------------------
c
c time measured in process time since initialization
c
c  seconds  reductions  redundant irredundant
c         MB    restarts       trail    variables
c          level   conflicts       glue     remaining
c
c *  0.00  4  0 0   0    0    0  0% 0    3   2 100%
c l  0.00  4  0 0   0    0    0  0% 0    3   2 100%
c 1  0.00  4  0 0   0    0    0  0% 0    3   2 100%
c
c --- [ result ] -------------------------------------------------------
c
s SATISFIABLE
v -1 2 0
c
c --- [ run-time profiling ] -------------------------------------------
c
c process time taken by individual solving procedures
c (percentage relative to process time for solving)
c
c         0.00   28.65% parse
c         0.00   26.56% search
c         0.00   25.52% lucky
c         0.00    0.00% simplify
c   =================================
c         0.00   42.11% solve
c
c last line shows process time for solving
c (percentage relative to total process time)
c
c --- [ statistics ] ---------------------------------------------------
c
c lucky:                         1       100.00 %  of tried
c minimized:                     0         0.00 %  learned literals
c shrunken:                      0         0.00 %  learned literals
c minishrunken:                  0         0.00 %  learned literals
c propagations:                  0         0.00 M  per second
c trail reuses:                  0         0.00 %  of incremental calls
c
c seconds are measured in process time for solving
c
c --- [ resources ] ----------------------------------------------------
c
c total process time since initialization:         0.00    seconds
c total real time since initialization:            0.00    seconds
c maximum resident set size of process:            3.62    MB
c
c --- [ shutting down ] ------------------------------------------------
c
c exit 10
```

Figure 4.6: Output of CADICAL run on the Tie & Shirt example.

```
c LOG 0 set option 'set ("log", 1)' from '0'
c LOG 0 API call 'set ("log", 1)' succeeded
c LOG 0 API call 'set ("--log")' succeeded
c ----------------------------------------------------------------------------
c LOG 0 API call 'set ("tieandshirt.cnf")' started
c LOG 0 API call 'set ("tieandshirt.cnf")' failed
c --- [ banner ] -------------------------------------------------------------
c
c CaDiCaL SAT Solver
c Copyright (c) 2016-2023 A. Biere, M. Fleury, N. Froleyks, K. Fazekas, F. Pollitt
c
c Version 2.0.0-rc.5 a3ebcea9903f7b71d28fb5133d81d5592b05089a
c g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0 -Wall -Wextra -g -DLOGGING
c Sat Jan 27 08:01:38 CET 2024 Linux abryzen9 6.5.0-14-generic x86_64
c will not generate nor write DRAT proof
c
c --- [ parsing input ] ------------------------------------------------------
c
c reading DIMACS file from 'tieandshirt.cnf'
c ----------------------------------------------------------------------------
c LOG 0 API call 'read_dimacs ("tieandshirt.cnf")' started
c opening file to read 'tieandshirt.cnf'
c found 'p cnf 2 3' header
c ----------------------------------------------------------------------------
c LOG 0 API call 'reserve (2)' started
c LOG 0 API leaves state CONFIGURING
c LOG 0 API enters state STEADY
c LOG 0 initializing 2 internal variables from 1 to 2
c LOG 0 enlarge internal size from 0 to new size 3
c LOG 0 initializing VMTF queue from 1 to 2
c LOG 0 queue unassigned now 1 bumped 1
c LOG 0 queue unassigned now 2 bumped 2
c LOG 0 initializing EVSIDS scores from 1 to 2
c LOG 0 finished initializing 2 internal variables
c LOG 0 enlarge external size from 0 to new size 3
c LOG 0 initialized 2 external variables
c LOG 0 mapping external 1 to internal 1
c LOG 0 mapping external 2 to internal 2
c LOG 0 API call 'reserve (2)' succeeded
c LOG 0 reserving 3 ids
c ----------------------------------------------------------------------------
c LOG 0 API call 'add (-1)' started
c LOG 0 activate 1 previously unused
c LOG 0 adding external -1 as internal -1
c LOG 0 API enters state ADDING
c LOG 0 API call 'add (-1)' succeeded
c ----------------------------------------------------------------------------
c LOG 0 API call 'add (2)' started
c LOG 0 API enters state STEADY
c LOG 0 activate 2 previously unused
c LOG 0 adding external 2 as internal 2
c LOG 0 API enters state ADDING
c LOG 0 API call 'add (2)' succeeded
c ----------------------------------------------------------------------------
c LOG 0 API call 'add (0)' started
c LOG 0 API enters state STEADY
c LOG 0 original clause -1 2
c LOG 0 new pointer 0x55f329eac4b0 irredundant size 2 clause[4] -1 2
c LOG 0 marking added irredundant size 2 clause[4] -1 2
c LOG 0 watch -1 blit 2 in irredundant size 2 clause[1] -1 2
c LOG 0 watch 2 blit -1 in irredundant size 2 clause[1] -1 2
c LOG 0 API call 'add (0)' succeeded
...
```

Figure 4.7: Partial output of CADICAL with logging run on the Tie & Shirt example.

```
int Internal::cdcl_loop_with_inprocessing () {
  int res = 0;
  ...
  while (!res) {
    if (unsat)
      res = 20;
    else if (unsat_constraint)
      res = 20;
    else if (!propagate ())
      analyze (); // propagate and analyze
    else if (iterating)
      iterate (); // report learned unit
    else if (!external_propagate ()) {
      if (unsat) res = 20;
      else analyze ();
    } else if (satisfied ()) { // found model
      if (!external_check_solution ()) {
        if (unsat) res = 20;
        else analyze ();
      } else if (satisfied ())
        res = 10; // model accepted by ExternalPropagator
    } else if (search_limits_hit ())
      break;      // decision or conflict limit
    else if (terminated_asynchronously ())
      break;      // externally terminated
    else if (restarting ())
      restart (); // restart by backtracking
    else if (rephasing ())
      rephase (); // reset variable phases
    else if (reducing ())
      reduce ();  // collect useless clauses
    else if (probing ())
      probe ();   // failed literal probing
    else if (subsuming ())
      subsume (); // subsumption algorithm
    else if (eliminating ())
      elim ();    // variable elimination
    else if (compacting ())
      compact (); // collect variables
    else if (conditioning ())
      condition (); // globally blocked clauses
    else
      res = decide (); // next decision
  }
  ...
  return res;
}
```

Figure 4.8: The main steps of the `cdcl_loop_with_inprocessing` function in CaDiCaL.

Chapter 5

Single Clause Assumption without Activation Literals to Speed-up IC3

Published In 2021 Formal Methods in Computer Aided Design (FMCAD)

Authors Nils Froleyks and Armin Biere

Changes from Published Version After the submission, Alexander Ivrii spotted a bug in the pseudo-code when comparing it to the implementation in CaDiCaL. This bug is fixed in Figure 5.2. The figure further includes an optimization implemented after the publication of this paper. See the caption of Figure 5.2 for more details.

Authors Contributions The author developed the idea for the extension together with A. Biere. He implemented it in CaDiCaL, including the deferred model reconstruction. He further modified ABC to use CaDiCaL. He ran and interpreted the presented evaluation. A. Biere extended the implementation at a later point to incorporate decision heuristics in the constraint literal selection.

Acknowledgement This work is supported by the Austrian Science Fund (FWF) under projects W1255-N23 and S11408-N23 as well as the LIT AI Lab funded by the State of Upper Austria.

Abstract We extend the well-established assumption-based interface of incremental SAT solvers to clauses, allowing the addition of a temporary clause that has the same lifespan as literal assumptions. Our approach is efficient and easy to implement in modern CDCL-based solvers. Compared to previous approaches, it does not come with any memory overhead and does not slow down the solver due to disabled activation literals, thus eliminating the need for algorithms like IC3 to restart the SAT solver. All clauses learned under literal and clause assumptions are safe to keep and not implicitly invalidated for containing an activation literal. These changes increase the quality of learned clauses, resulting in better generalization for IC3. We implement the extension in the SAT solver CaDiCaL and evaluate it with the IC3 implementation in the model checker ABC. Our experiments on the benchmarks from a recent hardware model

checking competition show a speedup for the average SAT call and a reduction in number of calls per verification instance, resulting in a substantial improvement in model checking time.

5.1 Introduction

Modern SAT solving is based on Conflict-Driven Clause Learning (CDCL) [128]. Many applications require solving a sequence of related SAT problems incrementally [68, 100], making use of inprocessing techniques [61, 62, 105] that make modern SAT solvers so efficient. Among those applications is the symbolic model checking algorithm IC3. In contrast to other incremental SAT-based techniques, such as bounded model checking (BMC) [48, 166] and k-induction [35, 167], IC3 does not rely on unrolling the transition function. As a result the SAT queries that IC3 poses are significantly smaller and faster to solve. However, the number of queries that IC3 makes over the course of one model checking procedure is significantly higher. We illustrate the kind of queries that IC3 makes in the following example.

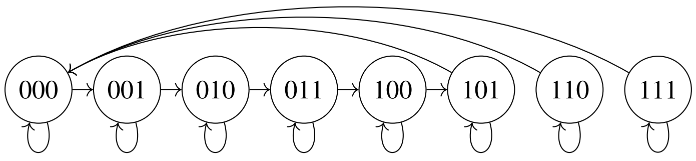

Figure 5.1: Transition system

Consider the transition system of a three-bit ($b_2 b_1 b_0$) counter, encoding integers up to seven, in Fig. 5.1. Non-deterministically, the counter is incremented, remains unchanged or is reset to zero after reaching five. Suppose we want to ensure that starting at state zero, all states with values greater than five are unreachable. A typical query asks "is state six reachable from any other state?", expressed as $SAT?[T \wedge (\neg b_2 \vee \neg b_1 \vee b_0) \wedge b'_2 \wedge b'_1 \wedge \neg b'_0]$, where T encodes the transition system for one step from $b_2 b_1 b_0$ to $b'_2 b'_1 b'_0$. It is unsatisfiable, telling us that state six is in fact unreachable. We can try to generalize this result to a set of states by considering a *cube* – an assignment to a subset of variables. The query $SAT?[T \wedge (\neg b_1 \vee b_0) \wedge b'_1 \wedge \neg b'_0]$ is satisfiable because state two can be reached from state one and $SAT?[T \wedge (\neg b_2 \vee b_0) \wedge b'_2 \wedge \neg b'_0]$ is satisfiable due to the transition from state three to state four. However, the query $SAT?[T \wedge (\neg b_2 \vee \neg b_1) \wedge b'_2 \wedge b'_1]$ is unsatisfiable, allowing us to conclude that all states in the cube $b_2 \wedge b_1$ are not reachable from outside the cube. We can use that insight to strengthen T by adding $\neg b'_2 \vee \neg b'_1$ to all future queries. This is in contrast to the clauses we previously added for only one query.

The popular assumption-based interface pioneered by MiniSat [48, 68] allows the user to specify a set of literals that are assumed to be true and picked by the solver as the first decisions. This allows us to add the assumption that a state is within a certain cube after the transition ($b'_2 \land b'_1$), however we still need to assume an additional clause encoding that the state is currently not within said cube ($\neg b_2 \lor \neg b_1$). The most common way to implement clause assumption, is to simulate the desired behavior using activation literals [48, 58]. Let C be a clause to add temporarily and a, the activation literal, a free variable, *i.e.,* it does not occur in the formula. By adding $C \lor a$ to the formula and assuming $\neg a$, we achieve the same as adding C to the formula. After a solution is found, the clause a is added, effectively removing C from the formula.

The problem with IC3 specifically, is the large number of queries made over the course of a single verification procedure. After a few hundred calls the activation literals clutter up the variable space and slow down the SAT solvers propagation. The common solution to this problem is to fully restart the SAT solver by replacing it with a fresh instance periodically, thus also deleting all learned clauses and heuristic scores. How to schedule these restarts in IC3 specifically, has been the topic of a full journal paper [168]. Using the technique presented in this paper, restarts are not necessary at all. Additionally learned clauses are safe to keep and will not contain an activation literal, which would make them useless for future calls.

Other approaches to clause assumption have been explored: Satire [63] supports pseudo-Boolean and other constraints. It records the dependencies of learned constraints explicitly, thus allowing the deletion of arbitrary clauses. In the SMT community, an interface based on pushing and popping on the assertion stack is prevalent [169]. Since constraints are removed in order, it is possible to mark a point in the data structures that maintain learned knowledge and remove everything past it, when a pop operation is executed. The first implementation of IC3 [49] used the SAT solver Zchaff [170]. It assigns an additional 32-bit integer to each clause. When learning a clause the bits of all dependencies are combined. The user can delete a group of clauses with a certain bit. This approach mostly simulates the use of activation literals and comes with a significant memory overhead.

This paper presents an extension of the prevalent assumption mechanism to additionally allow the assumption of a single clause, called *constraint* in the following. The extension can be implemented by a simple modification to the decision mechanism in a CDCL-based SAT solver. We implemented it in under 100 lines of code in the state-of-the-art SAT solver CaDiCaL. To evaluate our implementation we modify the IC3 engine in the model checker ABC to use CaDiCaL and clause assumption. As a first result, the changes simplify SAT solver usage and eliminate the need for restarts as well as some book-keeping for activation literals. An empirical evaluation on the 2019 hardware model checking competition [171] benchmark set shows that ABC spends less time outside of computing SAT queries, the number of queries per verification is reduced and the average SAT call is faster. Overall using clause assumptions yields a substantial speedup in verification time.

5.2 Incremental SAT and IC3

An *incremental* SAT solver solves a series of related formulas efficiently. It communicates with an application integrating it through an *interface* such as IPASIR [58]. It is implemented by all solvers participating in the incremental library track of the SAT Competition since 2015. The popular solver MiniSat along with all of its incremental descendants implement something very similar. We describe the relevant subset:

- `add(lit)` Add a literal to the current clause or if it equals 0, add the clause to the formula.
- `assume(lit)` Assume the literal to be true for the next solving attempt.
- `solve()` Return SAT if an assignment exists satisfying the formula and all assumptions, otherwise UNSAT.
- `val(lit)` Valid in SAT-case. Return the truth value of a literal in the satisfying assignment.
- `failed(lit)` Valid in UNSAT-case. Return *true* if the literal was assumed and used to prove unsatisfiability.

A prominent applications of incremental SAT-solving is the symbolic model checking algorithm IC3 by Bradley [49]. Given a transition system and a property P, IC3 tries to prove that it is not possible to reach a state that violates the property. It maintains a sequence of *frames* $F_0, F_1, \dots F_k$, each frame F_i is a formula encoding an overapproximation of the set of states reachable in at most i steps. The frames are refined by adding additional clauses until one of the frames contains all reachable states and none violates the property or a counterexample is found. Each frame has its own SAT solver instance that is initialized with an encoding of the transition function and updated with the new frame clauses.

The solvers are used almost exclusively to answer queries for predecessors of the form $SAT?[T \wedge F_i \wedge \neg s \wedge s']$, where T is the transition function and s is a cube. To refine the frames, a state s in the last frame that violates the property is identified with the query $SAT?[F_k \wedge \neg P]$. If no such state exists, a new frame is appended, otherwise IC3 tries to prove that the state is not actually reachable. The frames are queried for predecessors until an initial state is reached, thus producing a counterexample, or one of the frames returns unsat. In the latter case `failed` can be used to generalize the unreachable state to a cube, the negation of which is added to the frame. IC3 is guaranteed to eventually terminate with two consecutive frames containing the same set of states.

5.3 Assuming Clauses

Our main contribution is an extension to incremental SAT solvers that allows the assumption of an additional clause, called *constraint*, which is only valid during the next satisfiability query. Two functions are added to the interface:

- `constrain(lit)` Adds a literal to constraint. If a finalized constraint exists, delete it. If the literal equals zero, finalizes the current constraint.

- `constraint_failed()` Valid in *UNSAT* case. Return whether constraint was used to prove unsatisfiability.

Our approach is similar to the idea of model elimination [172]. We modify the decision heuristic to restrict the search to assignments that satisfy the constraint. The modified decision procedure is outlined in Fig. 5.2. The function decide is called initially at decision level 0. Decisions assigned to the trail are propagated outside of the function to assign truth values. Whenever a conflict arises, the decision level decreases and the assignments are backtracked [128]. Every assumption has a fixed decision level. In the case where an assumption is already satisfied, a *pseudo* decision level is introduced. Otherwise if an assumed literal is assigned to false at this point, the assignment is the result of propagating other assumptions together with original or learned clauses. Therefore the formula is unsatisfiable under the current assumptions if line 4 is reached.

At the first decision level after all assumptions have been assigned, three cases need to be considered: if one of the literals in the constraint is already satisfied, the search is not restricted. Otherwise one of the literals is picked as a decision to satisfy the constraint. In line 13 a variable selection heuristic can be used to pick the most promising literals first, similarly to [173, 174]. In the case where all literals are assigned to false, they are implied by the assumptions, thus cannot be assigned differently. The formula is therefore declared unsatisfiable under the assumptions and the constraint. This might only happen after additional clauses have been learned.

This approach to handle assumptions was pioneered by MiniSat [68]. It has been improved upon by collectively propagating the assumptions, using trail saving between incremental calls [175] or factoring out assumptions [176]. These techniques can be combined with the presented constraint mechanism.

Modern SAT solvers not only report unsatisfiability as a result, but also allow the user to query whether a particular assumption failed, *i.e.,* was used to prove unsatisfiability. This concept, introduced as `analyzeFinal` by MiniSat [68], is essential for the efficiency of many applications. If an original or learned clause is inconsistent with the assumptions, the last assumption picked as a decision is already assigned to false. Using a simple breadth-first search, the reasons for this assignment can be traced back through the implication graph [128]. The assumptions at the leaves of the search tree are marked as failed. In line 16, a similar search is initialized with the negation of every literal in the constraint. Thus, all assumptions necessary to prove unsatisfiability of the constraint in conjunction with the formula are marked as failed.

5.4 Experiments

We implemented the constraint interface in CaDiCaL [70] version 1.3.1. To increase confidence in the correctness of the SAT solver and its new extension, we used the model-based tester [150] that is integrated with CaDiCaL. It generates random sequences of

```
decide ( )

1  if level < |assumptions|
2      $\ell \leftarrow$ assumptions[level]
3      if val($\ell$) = false
4          analyzeFinal()
5      else if val($\ell$) = true
6          level++ // pseudo decision level
7      else trail[level++] $\leftarrow \ell$
8  else if level = |assumptions|
9      unassignedLit $\leftarrow$ 0
10     for $\ell$ in constraint
11         if val($\ell$) = false continue
12         if val($\ell$) = true
13             constraint.move_to_front($\ell$) // sorts over time
14             level++ // pseudo decision level
15             return
16         assert val($\ell$) = unassigned
17         if unassignedLit = 0 or better_decision($\ell$, unassignedLit)
18             unassignedLit $\leftarrow \ell$
19     if unassignedLit = 0
20         analyzeFinalConstraint() // cannot be satisfied
21     else trail[level++] $\leftarrow$ unassignedLit
22 else
23     $\ell \leftarrow$ literalSelectionHeuristic()
24     trail[level++] $\leftarrow \ell$
```

Figure 5.2: Algorithm decide picks the next decision to propagate.
Note that this version is modified from the one presented in the original paper:
1) Line 15 was missing. Thanks to Alexander Ivrii for spotting that.
2) In the version presented in the paper we did not use decision heuristics, i.e., better_decision always returned true.
3) We did not include the optimization in line 13.

API calls including assumptions and constraints together with random configurations for the solver. The returned models and failed assumption sets are checked for correctness. We ran the tester on 8 cores for multiple days to validate 1.2 billion test runs.

To evaluate our approach, we integrated CaDiCaL into the bit-level model checker

ABC[1] [177], replacing the integrated version of MiniSat [68]. There are two places where activation literals are used in ABC. The first is an alternative implementation of cube generalization, that is not used in the default configuration. In fact, it seems to not work correctly in the default version of ABC[1]. The other usage of activation literals is in the function that implements the predecessor query $SAT?[T \wedge F_i \wedge \neg s \wedge s']$. The transition function T and the frame F_i will only be extended with additional clauses, the cube s however changes at each query. The next-step cube s' is in conjunction with the rest of the formula and therefore translates to a set of unit clauses that can be implemented with assumptions. To combat the slowdown due to unused activation literals cluttering up the variable space, ABC replaces the SAT solver with a new instance after adding 300 activation literals. Using the extended interface, the negated cube $\neg s$ can be added as a constraint, thus eliminating the restarts.

We tested five configurations: the original version of ABC (Og), disabled SAT solver restarts (Di), a version with CaDiCaL as backend using activation literals (Ca) and one also using CaDiCaL but the new constraint interface instead of activation literals (Co). As an additional result we present a slight modification to the last configuration that defers model reconstruction [62] in the SAT-case and failed literal collection in the UNSAT-case until a model or a failed literal is queried respectively (De). Using a heuristic to pick the literals from the constraint has not been successful. ABC uses a priority metric to order the literals of the cube s by default. Using this order for the constraint turned out to be superior to the heuristics available in CaDiCaL.

Our evaluation follows the principles laid out in SAT manifesto v1.0. [178]. The source code used for the evaluation and the generated log files are available on our website[2]. The experiments are run in parallel on 32 nodes of our cluster. Each node has access to two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled) and 128 GB main memory. We allocate 4 instances of ABC to every node. The time limit is set to 1 hour of wall-clock time, memory is limited to 30GB per instance. The memory limit is the only aspect that differs from the setup used in the hardware model checking competition. However, the maximum memory consumption was observed to be below 1.5GB.

The evaluation is based on the benchmark set used in the 2019 model checking competition [171]. It contains 219 instances, 15 of which we removed because they were not solved by any tested configuration. We use PAR-2 scoring to compare the configurations. PAR-2 assigns the runtime in seconds or twice the time limit (7200) if an instance was not solved. The other columns list additional measurements for the two configurations using CaDiCaL, one with activation literals (Ca) and the other using constraints instead (Co). The number of restarts is zero if constraints are used and therefore not shown. Besides that, we list the number of SAT calls (in thousands), along with the average time per call in milliseconds. We display the measured data for instances, where at least one configuration took more than two seconds, along with an average over all 204 instances. The results are presented in Table 5.1.

[1]commit f87c8b4
[2]http://fmv.jku.at/assumingclauses

Table 5.1: Experimental results.

	PAR-2					Res.	Calls		TpC	
	Di	Og	Ca	Co	De	Ca	Ca	Co	Ca	Co
Mean	80	46	16	8.93	**8.21**	61	19	**15**	0.61	**0.51**
beemTele6Int	136	7200	**53**	181	101	520	**157**	574	**0.24**	0.27
toyLock4	7200	483	1731	**357**	359	7459	2251	**1098**	0.42	**0.25**
visArraysField5	7200	1.6	**0.58**	51	34	1	**1**	113	0.53	**0.41**
nan	208	421	163	158	**140**	1381	**420**	423	**0.29**	0.32
beemColl6Int	241	258	322	133	**108**	398	123	**91**	2.31	**1.24**
cal110	213	168	130	**110**	122	191	59	**42**	**1.96**	2.39
cal109	179	197	102	117	**86**	110	**34**	44	2.71	**2.44**
cal93	186	136	121	**118**	140	206	63	**58**	**1.69**	1.8
cal94	127	160	115	**95**	131	171	52	**41**	**1.94**	2.1
cal100	112	**42**	67	67	54	148	45	**44**	**1.23**	1.29
cal131	46	**44**	77	58	60	136	42	**35**	1.58	**1.41**
cal146	47	39	71	42	**38**	131	41	**23**	**1.51**	1.55
cal136	**34**	46	59	43	35	100	31	**23**	1.62	**1.59**
cal128	52	38	46	**37**	40	99	31	**25**	1.29	**1.27**
beemExit5Int	51	17	26	16	**15**	357	110	**86**	0.18	**0.15**
cal134	38	47	50	48	**36**	79	**25**	26	1.72	**1.57**
cal132	39	36	48	42	**32**	83	26	**24**	1.57	**1.54**
cal144	**30**	34	41	33	42	64	20	**17**	1.7	**1.64**
beemLampNat5Int	26	23	**23**	35	31	193	**61**	102	**0.28**	0.3
cal89	16	**14**	32	33	25	68	22	**18**	**1.23**	1.6
beemRether4Bstep	13	**4.29**	16	7.16	6.99	91	29	**13**	**0.42**	0.49
beemBrp2Int	16	5.1	3.6	0.76	**0.74**	86	29	**7**	0.08	**0.07**
beemFrogs2Bstep	**2.47**	2.53	12	5.59	4.74	31	10	**4**	**1.12**	1.27
beemAdding5Int	1.78	3.9	2.07	1.12	**1.09**	53	17	**11**	0.08	**0.07**
visArraysTwo	1.35	2.89	3.89	0.57	**0.55**	99	30	**5**	0.09	**0.07**
Heap	2.02	1.9	3.38	1.68	**1.63**	57	22	**13**	0.11	**0.09**

Disable restarts, **O**ri**g**inal version of ABC, **Ca**DiCaL backend, **Co**nstraint interface used, **De**fer model reconstruction

Comparing the first two columns, it is evident that if activation literals are used, solver restarts are necessary. It has been suggested [168] that because the queries posed by IC3 are small but numerous, IC3 implementations should prefer faster SAT solvers to more powerful ones. Comparing the original with the CaDiCaL version shows that while using MiniSat is faster on a number of instances, using CaDiCaL seems to be an advantage on the harder instances. In fact, using the newer SAT solver, one additional instance can be verified. Over all instances a speedup of 2.82 is observed.

With the version using CaDiCaL and activation literals as a baseline, we observe a speedup of 1.84 when switching to constraints. The time spend outside the SAT solver is reduced to below 20%, by eliminating the actual SAT solver restarts and the repeated loading of the transition relation [179]. Beyond that, the average SAT call is 16% faster. This can partially be explained by the solver not being slowed down by activation literals. We conjecture that, more importantly, the "quality" of the learned clauses in the solvers database is higher. Since clauses are not deleted by restarts and none of the learned clauses are implicitly disabled for containing an activation literal, the solver can profit from shorter and more useful clauses. Measuring this quality however, is outside the scope of this paper. An additional effect is that these clauses allow conflicts earlier in the search tree, resulting in fewer failed literals and thus allows for better generalization in IC3. This can explain why 21% fewer calls are made.

The last two columns listing PAR-2 scores reflect small changes in the solver. Deferring the model reconstruction results in an additional speedup of 9%, increasing the total speedup compared to the original version to 5.64.

5.5 Conclusion

We present a simple extension to the commonly used incremental SAT solver interface IPASIR that simplifies solver usage and is easy to implement by modern SAT solvers. The extension gives an alternative to the techniques described in the journal paper [168] and partially implemented in ABC. Our experiments using the new technique with ABC show a substantial improvement in model checking time. Compared to the original IC3 engine, our final implementation is more than five times faster.

Since the development of new SAT encodings depends on the interface available, we hope that new applications will arise as more solvers implement the extended interface presented in this paper.

Handling more than one constraint can be achieved by using a complete model elimination search over the constraints. This would however increase the implementation effort. Since inprocessing techniques cannot be applied, model elimination might be less effective than using activation literals, if the number of temporary clauses is high. We leave this investigation to future work.

Chapter 6

CadiBack: Extracting Backbones with CaDiCaL

Published In 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)

Authors Armin Biere, Nils Froleyks, and Wenxi Wang

Changes from Published Version Added additional plots to the Appendix (6.2, 6.3, and the 2020 and 2021 plots in Figure 6.5). Changed layout of the pseudo-code.

Authors Contributions
The pseudo-code presented in this paper was written by the author. The additional ideas, particularly those related to the use of constraints—crucial for performance—emerged from discussions with A. Biere. A. Biere implemented the initial version of CadiBack used in this paper. The evaluation was carried out jointly.

Abstract The backbone of a satisfiable formula is the set of literals that are true in all its satisfying assignments. Backbone computation can improve a wide range of SAT-based applications, such as verification, fault localization and product configuration. In this tool paper, we introduce a new backbone extraction tool called CADIBACK. It takes advantage of unique features available in our state-of-the-art SAT solver CADICAL including transparent inprocessing and single clause assumptions, which have not been evaluated in this context before. In addition, CADICAL is enhanced with an improved algorithm to support model rotation by utilizing watched literal data structures. In our comprehensive experiments with a large number of benchmarks, CADIBACK solves 60% more instances than the state-of-the-art backbone extraction tool MINIBONES. Our tool is thoroughly tested with fuzzing, internal correctness checking and cross-checking on a large benchmark set. It is publicly available as open source, well documented and easy to extend.

6.1 Introduction

In 1997, Parkes first defined the backbone of a propositional formula as the set of literals whose assignments are true in every satisfying assignment [180]. The size of the backbone is associated with the hardness of the corresponding propositional problem [181, 182]. Usually, the larger a backbone, the more tightly constrained the problem becomes, thus the harder for the solver to find a satisfying assignment [183, 184]. It is proved by Janota that deciding if a literal is in the backbone of a formula is co-NP complete [185]. Furthermore, Kilby et al. show that even approximating the backbone is intractable in general [186].

Nevertheless, the identification of the backbone (either in a partial or a complete way) has a number of practical applications, such as post-silicon fault localization in integrated circuits [187–189], interactive product configuration [185], facilitating the solving efficiency of MaxSAT [190–193] and random 3-SAT problems [194], as well as improving the performance of chip verification [195]. Motivated by the wide range of applications, developing efficient algorithms for computing the backbone of a given propositional formula is important.

Indeed, numerous techniques to compute the backbone have been proposed during the past few decades. These approaches make use of four main techniques: (i) model enumeration, which enumerates all models of a satisfiable formula to identify the backbone; (ii) iterative SAT testing, which repeatedly filters out a candidate or include it in the backbone; (iii) upper bound checks, which try to identify multiple backbone literals at once; and (iv) the core-based method, which is guided by unsatisfiable cores and tries to eliminate as many candidates at once as possible. For example, Kaiser et al. [196] designed three model-enumeration algorithms. Climer et al. [197] propose a graph-based iterative SAT testing approach.

Later, Zhu et al. [187, 188] designed more efficient SAT testing approaches for post-silicon fault localization. Note that, the backbone extractor MINIBONES [145, 198] implements both an iterative and a core-based approach. Despite recent attempts [199–201] to improve upon MINIBONES, the corresponding tools are not publicly available, and no significant advances have been made so far which still leaves MINIBONES as the state-of-the-art.

Our new backbone extractor CADIBACK improves the iterative algorithms of MINIBONES [145, 198] and uses the state-of-the-art SAT solver CADICAL [73], extended with new flipping algorithms to support backbone extraction. Different configurations of these algorithms are implemented inside CADIBACK, empirically evaluated and compared with MINIBONES on a large set of satisfiable instances collected from the SAT Competitions from 2004–2022, on which CADIBACK solves 60% more instances.

The paper is structured as follows. After this introduction, we discuss basic concepts and notations related to backbone extraction in Section 6.2. The relevant backbone extraction algorithms of MINIBONES are introduced in Section 6.3. We then present our improvements over these algorithms and propose CADIBACK in Section 6.4. The implementation details of CADIBACK are provided in Section 6.5. Finally, we empirically evaluate CADIBACK in Section 6.6 and draw conclusions in Section 6.7.

6.2 Basic Concepts and Notations

Consider a propositional *formula* φ in conjunctive normal form (CNF) over a fixed set of variables $\mathcal{V}$ and literals $\mathcal{L} = \mathcal{V} \cup \overline{\mathcal{V}}$, where $\overline{\mathcal{V}} = \{\bar{v} \mid v \in \mathcal{V}\}$ denotes the negated variables. For a literal $\ell \in \mathcal{L}$, we define $v = |\ell|$ as the variable of ℓ, i.e., $\ell \in \{v, \bar{v}\}$. In this paper, we mainly consider full *assignments* $\sigma \colon \mathcal{V} \to \{0, 1\}$ assigning variables to Boolean constants "0" (*false*) or "1" (*true*). For convenience, we use the set and logic notation interchangeably for formulas φ, clauses $C \in \varphi$ and literals $\ell \in C$, as well as assignments $\{\ell \mid \sigma(\ell) = 1\}$. The notion of assignments is lifted to literals, formulas and clauses in the natural way through substitution followed by Boolean expression simplification. A *model* of φ is an assignment σ with $\sigma(\varphi) = 1$ and also called satisfying assignment. A formula is *satisfiable* if it has a model. Otherwise, it is *unsatisfiable*. In this paper, we focus on satisfiable formulas φ.

A literal ℓ is a *backbone literal* of a formula φ iff there exists a model σ of φ with $\sigma(\ell) = 1$ and all other assignments σ' with $\sigma'(\ell) = 0$ do not satisfy φ, i.e., $\sigma'(\varphi) = 0$. The *backbone* $\mathcal{B}$ of a formula φ is the set of its backbone literals. We introduce two conditions that determine whether literals are included or not in the backbone $\mathcal{B}$.

The first condition is based on identifying *fixed* literals. A clause $C = \ell$ (or $C = \{\ell\}$ in set notation) having a single literal ℓ is called *unit* clause. If a unit clause $C \in \varphi$, the corresponding literal ℓ is clearly a backbone literal. We call such literals *fixed*. This also applies to unit clauses deduced by the SAT solver through for instance clause learning, simplification and preprocessing [60]. All such fixed literals are included in the backbone $\mathcal{B}$.

The second condition is called *disagreement condition*, stating that if there are two models σ and σ' *disagreeing* on ℓ, i.e., $\sigma'(\ell) = \overline{\sigma(\ell)}$, then neither ℓ nor its negation are backbone literals (i.e., $\ell, \bar{\ell} \notin \mathcal{B}$). This can be realized by using each newly discovered model σ' to *filter* the list of remaining backbone candidates. For instance, the empty formula over n variables has both constant assignments $\sigma \equiv 0$ and $\sigma' \equiv 1$ as models, disagreeing on all literals, and thus $\mathcal{B} = \emptyset$. Note that, there is a special case of the disagreement condition called *model rotation*, as described in [145]. Similar ideas have been used for MUS extraction [146]. The literal ℓ is *rotatable* [145] in a model σ of φ iff $\sigma(\ell) = 1$ and the assignment τ that differs from σ only in $|\ell|$ is a model of φ (τ can be taken as the special case of σ' in the disagreement condition). We also call such literals *flippable*, which applies for the rest of the paper.

Obviously, a literal which can be flipped is not a backbone literal, nor is its negation, and both can be dropped from the backbone candidate list. Example 6.1 below shows how a literal is determined to be flippable under the model rotation condition.

Example 6.1. Consider $\varphi = (\bar{c} \vee t) \wedge (c \vee e) \wedge \varphi'$ which encodes "if-then-else(c, t, e)", where neither c nor $\bar{c}$ occur in φ' but e and t do (they are not "pure"). Assume that the constant true assignment $\sigma \equiv 1$ is a model, i.e., $\sigma(\varphi') = 1$. Both t and e are set to true, but only the literal c can be flipped. In the resulting model τ, all variables are set to true except for c, and $\bar{c}$ can be flipped (back) in τ to obtain the original model σ. Thus, literal c is flippable.

6.3 Algorithms in MINIBONES

The backbone extraction algorithms of MINIBONES [145] take advantage of incremental SAT solving (refer to [5, 62, 68] for details) to gradually augment the original formula with implied clauses (particularly learned clauses). These clauses are added implicitly to the single SAT solver instance during incremental queries, while assuming the negation of one or more remaining backbone candidate literals. Specifically, these iterative MINIBONES algorithms (Algorithms 3, 4 and 5 in [145]) utilize discovered models and model rotation to refine the set of candidate literals $\Lambda \subseteq \mathcal{L}$ which is initialized as $\Lambda = \{\ell \mid \sigma(\ell) = 1\}$ by the first discovered model σ. On termination ($\Lambda = \emptyset$), the backbone $\mathcal{B}$ matches the fixed literals (of the augmented formula) and all other literals are dropped.

There are three iterative algorithms proposed for MINIBONES in [145]. The basic algorithm (Algorithm 3 in [145]) needs at least as many iterations as the number of backbone literals, which is inefficient on formulas with exactly one solution but many variables. An improved algorithm (Algorithm 4 in [145]) assumes that at least one of the remaining candidate literals can be flipped (i.e., using activation literals a temporary clause is added that contains the disjunction of the negated candidates). If the SAT query under such assumption is unsatisfiable, all candidates are fixed and the backbone extraction is done. A more advanced algorithm (Algorithm 5 in [145]) only adds a subset of the remaining candidates, called a *chunk*, to the temporary clause. Chunks are limited in size to avoid thrashing the SAT solver with too large temporary clauses and make it more likely for a call to be unsatisfiable.

Furthermore, MINIBONES proposes a new model rotation algorithm (Section 5 in [145]) to determine flippable (rotatable) literals based on the notion of *forcing*. A clause C *forces* a literal $\ell \in C$ under assignment σ, if $\sigma(C) = \sigma(\ell) = 1$ and $\tau(C) = 0$ with τ obtained from σ by flipping ℓ. A literal ℓ is *forced* in a formula φ under a model σ, if there is a clause $C \in \varphi$ which forces ℓ under σ. It is straightforward to see that literals which can be flipped in a model σ of φ are exactly those that are not forced. Based on this observation, the model rotation algorithm goes over all clauses whenever a new model is found and identifies literals that are not forced by any of them. If any of the remaining backbone candidates are not forced, they are dropped from the candidate list.

6.4 Improved Algorithms in CADIBACK

Algorithm 1 Extracting backbone of formula φ.

```
// Assume $\varphi$ is satisfiable and use
// $K = 1$   for one-by-one,
// $K = 10$ for chunking and
// $K = \infty$ as default (non-chunking).
backbone (CNF $\varphi$,  chunk rate $K = \infty$)
1   $(res, \sigma) \leftarrow$ SAT$(\varphi)$
2   assert $\sigma(\varphi) = res = 1$                // $\varphi$ satisfiable!
3   $\Lambda \leftarrow \{\ell \in \sigma \mid \neg$flippable$(\ell, \sigma)\}$   // candidates
4   $k \leftarrow 1, \quad \mathcal{B} \leftarrow \emptyset$
5   while $\Lambda \neq \emptyset$ do
        // $F \leftarrow \emptyset$ for no-fixed, otherwise by default
6       $F \leftarrow \{\ell \in \Lambda \mid \ell$ is fixed by SAT in $\varphi\}$
7       $\mathcal{B} \leftarrow \mathcal{B} \cup F, \quad \Lambda \leftarrow \Lambda \setminus F$
8       $\Gamma \leftarrow$ pick $k'$ literals from $\Lambda$       // chunk
                with $k' = \min(k, |\Lambda|)$
9       $\rho \leftarrow \bigvee_{\ell \in \Gamma} \bar{\ell}$                 // constraint: flip one in chunk
        // Solve $\varphi$ under $\rho$ with "bool constrain"
        // or use activation literal for no-constrain.
10      $(res, \sigma) \leftarrow$ SAT$(\varphi \mid \rho)$
11      if $res$ then                         // SAT call satisfiable
            // filter only a single literal for no-filter
12          $\Lambda \leftarrow \{\ell \in \Lambda \mid \sigma(\ell)\}$
13          $\Lambda \leftarrow \{\ell \in \Lambda \mid \neg$flippable$(\ell, \sigma)\}$
14          $k \leftarrow 1$                        // reset chunk size to 1
15      else                                  // SAT call unsatisfiable
16          $\mathcal{B} \leftarrow \mathcal{B} \cup \Gamma$
17          $\Lambda \leftarrow \Lambda \setminus \Gamma$
18          $k \leftarrow K \cdot k$                    // increase size geometrically
19  return $\mathcal{B}$                          // or print when literal is added
```

Algorithm 2 Checking if literal ℓ
can be flipped in model σ.

```
// Assume $\sigma(\varphi) = \sigma(\ell) = 1$, unit clauses
// have exactly one watched literal
// and all other clauses are watched
// by two literals $w_1 \neq w_2$ with
// $\sigma(w_1) = 1$ if $\sigma(w_2) = 0$ and vice versa.
flippable (CNF $\varphi$, literal $\ell$, model $\sigma$)
  // return 0 for no-flip
  for all clauses $C$ watched by $\ell$ in $\varphi$
      if $\sigma(C \setminus \{\ell\}) = 0$ then return 0
  return 1
```

Algorithm 3 Picking the next decision literal under clausal constraint ρ and the partial model σ.

```
// Given a single clausal constraint
// $\rho = \ell_1 \vee \cdots \vee \ell_k$ and assignment $\sigma$
// determine whether $\rho$ is conflicting.
// Otherwise pick new decision.
decide (constraint $\rho$, partial model $\sigma$)
  ... // handle literal assumptions
  if $\sigma(\rho) = 1$ then              //constraint true
      $\ell \leftarrow$ "first" literal in $\rho$ with $\sigma(\ell)$
      // speed-up future search for $\ell$
      move $\ell$ to the front of $\rho$
  elif $\sigma(\rho) = 0$ then            //constraint false
      ... // handle conflicting constraint
  else                                    // constraint undetermined
      $\ell \leftarrow$ highest scored literal in $\sigma(\rho)$
      pick $\ell$ as new decision and return
  ... // fall back to default decisions
```

Our backbone algorithm combines all three iterative approaches from [145]. It simulates the basic iterative Algorithm 3 in [145] for $K = 1$ and comes close to the improved Algorithm 4 in [145] for $K > |\Lambda|$ and the most advanced Algorithm 5 in [145] for other values of K. The difference between our algorithm and the latter two is that we use a dynamic chunk size that is reset to 1 after a satisfiable call and grows geometrically as long SAT queries remain unsatisfiable. In any case, it first identifies an initial model σ and initializes the set of candidates Λ after filtering out flippable literals F. The remaining candidates are examined in chunks Γ. If all of the literals in the chunk are backbones, the chunk size is increased. Otherwise, the solver returns a new model σ which is used to filter the candidate list, as it is guaranteed to disagree with the previous model in at least one of the literals in the current chunk by assuming the constraint ρ. After that, another model rotation is performed and the chunk size is reset to 1. Note that, instead of including explicit insertions of backbones, we can assume that the SAT solver does the insertion implicitly. Flippable literals are identified by the new flippable algorithm which only traverses clauses watched by the remaining backbone candidates. The decide algorithm is an optimized version of the decision procedure in our SAT solver for more efficiently handling large constraints as they arise in this application, which picks the literal with the highest variables scores (i.e., EVSIDS scores [202] or VMTF stamps [203]).

CADIBACK is built upon the state-of-the-art SAT solver CADICAL [73] which has been extended with additional algorithms to support backbone extraction. The general backbone extraction algorithm of CADIBACK is shown in Algorithm 1. It follows the iterative algorithms of MINIBONES, which uses complements of backbone estimates (as constraints) and chunking, but with three key improvements.

First, CADIBACK uses transparent incremental inprocessing [62], as CADICAL is able to effectively and efficiently simplify the formula (e.g., using variable elimination) during incremental queries completely transparent to the user, while MINIBONES does not support inprocessing due to the limitation of its base solver MINISAT [68].

Second, to assume the disjunction of the complements, CADIBACK utilizes single clause assumptions through the "`void constrain (int lit)`" API call in CADICAL [5], instead of adding a clause with the complement literals and an activation literal [68], as in MINIBONES. The reason is that these added clauses and variables by MINIBONES may risk to clog the SAT solver, and handling constraints explicitly can have the benefit to give the SAT solver more control on selecting decisions. Since the assumed clausal constraint contains a high number of literals in this application ($|\mathcal{V}|$ initially), we extended the existing implementation of single clause assumptions in CADICAL slightly, as shown in Algorithm 3. After each restart the SAT solver is forced to decide on a literal to satisfy the constraint. CADIBACK chooses the one with the highest variable score (EVSIDS scores [202] or VMTF stamps [203]) among all unassigned literals in the constraint.

Third, while in earlier work model rotation only had negative effects on MINIBONES [145], we show that CADIBACK benefits from using model rotation to improve efficiency of backbone computation. The key of this improvement is our fast flipping algorithm implemented in CADICAL, accessible through the new API call "`bool flippable (int lit)`". As described in Algorithm 2, it uses watch lists to find individual "flippable" literals in models through propagation instead of going over the whole formula to find unit clauses. We also consider a variant of Algorithm 2 which eagerly flips flippable literals as they are found, with the goal to drop even more backbone candidates through flipping. The following example shows the possibility that flipping a flippable literal can yield additional flippable literals.

Example 6.2. Continuing Example 6.1, assume that no clause in φ' forces literal t under $\sigma \equiv 1$, which is not the case for the first clause $(\bar{c} \vee t)$ in φ, as it forces t under σ. Thus, t cannot be flipped in σ. As c does not occur in φ', there is no clause forcing $\bar{c}$ under τ. In addition, the only other clause $(\bar{c} \vee t)$ with t is not forcing as it is satisfied by two literals. Thus, flipping c makes t flippable in τ (τ' obtained from τ by flipping t remains a model of φ). Therefore, neither c nor t are backbone literals.

To implement this idea we provide a new "`bool flip (int lit)`" API call in CaDiCaL which implements a variant of Algorithm 2, inspired by propagation in SAT solvers. While for "`flippable`" we only need to check that there is another satisfied literal in all traversed clauses watched by the literal ℓ requested to be flipped, the "`flip`" implementation needs to unwatch ℓ in these clauses and watch that other satisfied literal instead (unless the second watched literal in the clause is also satisfied). If finding replacements is successful for all clauses watched by ℓ (or the other watched literal is satisfied), the value of ℓ is flipped. Otherwise, it remains unchanged and "`flip`" fails. Note that this variant of our flipping algorithm was previously implemented inside the sub-solver Kitten of Kissat to diversify models with the goal of speeding up the refinement process of SAT sweeping [144].

Algorithm 3 of [145] can be simulated precisely with our algorithm by setting $K = 1$. However, Algorithm 5 of [145], which uses a fixed chunk size limit can only be approximated by setting $K = 100$, as we change the chunk size k dynamically. Our adaptive scheme increases k geometrically with rate K as long as SAT queries remain unsatisfiable (which fixes all backbones in the chunk at once). If the SAT solver finds a model instead then the chunk size k is reset to one, i.e., the next constraint will only contain the negation of a single backbone candidate. With $K = \infty$ the SAT solver is either assuming the complement of a single or the disjunction of the negation of all remaining backbone candidates which is the setting of our algorithm closest to Algorithm 4 of [145] which does not limit chunk size at all.

6.5 Implementation

Our tool CadiBack uses the extended CaDiCaL [73] and is implemented in roughly 1200 lines of C++ code (counted after formatting with ClangFormat). The source code available at https://github.com/arminbiere/cadiback is concise and well-documented.

To check the correctness of algorithms and implementations, an internal backbone checker is implemented inside CadiBack. The checker can be enabled through the command line option "`--check`" and is simply a SAT solver instance of CaDiCaL, i.e., if checking is enabled, CadiBack obtains the checker instance as a copy of the main internal CaDiCaL solver through the "`copy`" API call provided by CaDiCaL.

First, it checks correctness of an identified backbone literal ℓ, by confirming that the input formula φ under the assumption $\neg\ell$ (negation of the backbone) is unsatisfiable. Second, it checks the correctness of dropping a literal ℓ from being a backbone candidate (removed from set Λ in Algorithm 1), by confirming that the input formula φ remains satisfiable under the assumption $\neg\ell$. Third, it checks whether the number of backbone literals found and the number of dropped literals sum up to the number of variables in the input formula.

Standard grammar-based black-box fuzz-testing was applied [158] with the backbone checking enabled on all 42 compatible pairs of options used in our experiments in Section 6.6. This pairwise combinatorial testing [204] through fuzzing was run for 50 hours in parallel using as many processes as configurations on a dual processor AMD EPYC 7343 machine (providing in total 64 virtual cores). In addition, sizes of the backbones of all our benchmarks (see Section 6.6) were sanity checked with the ones computed by 12 configurations of CADIBACK and two configurations of MINIBONES considered in our experiments.

In addition, for flipping information extraction, the library of CADICAL is extended to provide "`bool flippable (int)`" and "`bool flip (int)`" as discussed in the last section. The model based tester MOBICAL is also extended correspondingly for testing the new functionality. This extended version of CADICAL with improved constrain handling and flipping is also available at https://github.com/arminbiere/cadical.

6.6 Experiments

Benchmarks. To evaluate CADIBACK empirically, we collected all benchmarks from the main track of the SAT competition 2004 to 2022 as our initial benchmark set. We noticed that benchmarks from one competition year often contain old benchmarks (sometimes arbitrarily renamed or commented by the competition organizers) from previous competition years. This caused our initial benchmark set to include several redundant benchmarks. To remove such duplicates, in a second step, we cleaned up each individual benchmark by removing comments using a simple DIMACS pretty printer, followed by identifying identical benchmarks through computing an MD5 checksum and removing redundant ones. Then we ran the state-of-the-art SAT solver Kissat 3.0.0 [144] with 5,000 second timeout on the no-duplicate benchmark set and selected benchmarks solved to be satisfiable. In total this yields 1798 benchmarks available at https://cca.informatik.uni-freiburg.de/sc04to22sat.zip (6 GB) and [31].

Baseline. We choose the state-of-the-art backbone solver MINIBONES as our baseline (ported to support newer C++ compilers available at https://github.com/arminbiere/minibones). As suggested by [145], we use the configuration "`-e -c 100 -i`" (called minibones-core-based), which adopts the core-based approach with a fixed chunk size of 100 and inserts found backbone literals into the input formula explicitly. Additionally, to evaluate how our algorithm improves upon the Algorithm 5 in [145], we choose "`-u -c 100 -i`" (called minibones-iterative) which implements the algorithm and uses activation literals.

Platform. For benchmark collection we used a machine with an AMD Ryzen Threadripper 3970X 32-Core Processor at 4.5 GHz and 256 GB RAM. All other experiments were conducted in parallel on a cluster consisting of 32 machines, each with two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled) and 128 GB

RAM. Each instance is allocated to one core with a timeout of 5,000 seconds and a memory limit of 7 GB.

Data Availability. Experimental data including source code and log files are available on https://cca.informatik.uni-freiburg.de/cadiback.

Overall Results. We run both CADIBACK and the baseline MINIBONES on all benchmarks in our benchmark set. We consider an instance solved if the tool completes backbone computation, i.e., classifies all literals as either backbone or non-backbone. The number of instances solved over time are presented in Figure 6.1. It turns out that the best performing default configuration (default) of CADIBACK can solve 732 instances in total, which is 274 more (59.82%) than the best performing configuration of MINIBONES, i.e., the iterative configuration minibones-iterative, which solves only 458 instances. Note that, 11 failing runs of CADIBACK and 61 failing runs of MINIBONES hit the memory limit. It is also instructive to observe that over all selected 1798 instances CADIBACK was able to find the first model in 1573 cases, while MINIBONES did so in only 1152 cases, which clearly shows the advantage of using CADICAL [73] versus MINISAT [68] in this application. It might be interesting to investigate whether this improvement transfers to other applications using MINISAT.

In addition, following the SAT manifesto v1.0 [178], we also compare the default configuration of CADIBACK with the best configuration of MINIBONES on all satisfiable benchmarks in the main track of SAT competitions from 2020 to 2022. As a result, CADIBACK/MINIBONES solved 22/9 instances in 2020, 52/17 in 2021 and 41/10 in 2022.

Configurations. We study the impact of different design options in CADIBACK by evaluating 12 configurations (see Figure 6.1) including an extension implementing the core-based approach of MINIBONES (Algorithm 7 in [145]). Firstly, we observe that the effects of using smaller chunks were detrimental in our experiments. In fact, the infinite chunk size $K = \infty$ (default) has been very beneficial which solves 732 instances, while chunk size $K = 10$ (chunking) solves only 702 instances and chunk size $K = 1$ (one-by-one) even only 692.

Secondly, we study the impact of design options related to flipping. The experimental results indicate that removing flippable literals from the candidate list (no-flip) does not have a significant overall impact. This result differs from the one given by the authors of MINIBONES where the model rotation was detrimental. We attribute this to the efficiency of using watch lists for the flippable check. The really-flip configuration uses "`flip`" and simply tries to flip literals of the candidate chunk in an arbitrary order. It performs similar to the default configuration which uses "`flippable`", but is better in the aspect that the default only found 30,780,841 flippable literals in total, while really-flip found 32,488,468. This directly leads to a reduction of the total number of SAT solver calls, which goes down from 2,070,166 calls in no-flip to 1,478,160 calls in default and even down to 992,404 calls in really-flip.

Thirdly, to evaluate the impact of CADICAL on CADIBACK in detail, including its more advanced inprocessing and its most recent "`constrain`" API to support single clause assumptions [5]. We observe that disabling the inprocessing in CADICAL (no-inprocessing) significantly degrades the performance from solving 732 instances to 690 instances. Disabling the single clause assumption support from CaDiCaL in configuration no-constrain and falling back to activation literals (as MiniBones does) degrades the efficiency of CadiBack even more significantly to solving only 672 instances.

Lastly, we evaluate the impact of our core-based algorithm. The core-based preprocessing in CadiBack only solves 672 instances. However, since the core-based approach falls back to default if the considered literal set becomes empty after removing failed assumptions (see Algorithm 7 in [145] for details), our cores version is more sophisticated than MiniBones (minibones-core-based), thanks to its advanced features in default. In contrast, the core-based MiniBones configuration (minibones-core-based) is slightly better than its iterative version (minibones-iterative) for shorter run times, which matches observations of [145] with the lower time limit of 800 seconds. One can argue, that the reason probably is that the core-based algorithm in MiniBones can rely on literal assumptions [68] avoiding the overhead inflicted from adding temporary clauses and activation literals. However, this slight advantage degrades for long running instances, as can be seen in Figure 6.1.

6.7 Conclusion

We revisited backbone algorithms and implemented a new open-source backbone extraction tool CadiBack based on an extended version of the state-of-the-art SAT solver CaDiCaL. Our extensive evaluation on a large set of benchmarks shows a substantial performance improvement by solving 60% more benchmarks than the state-of-the-art MiniBones.

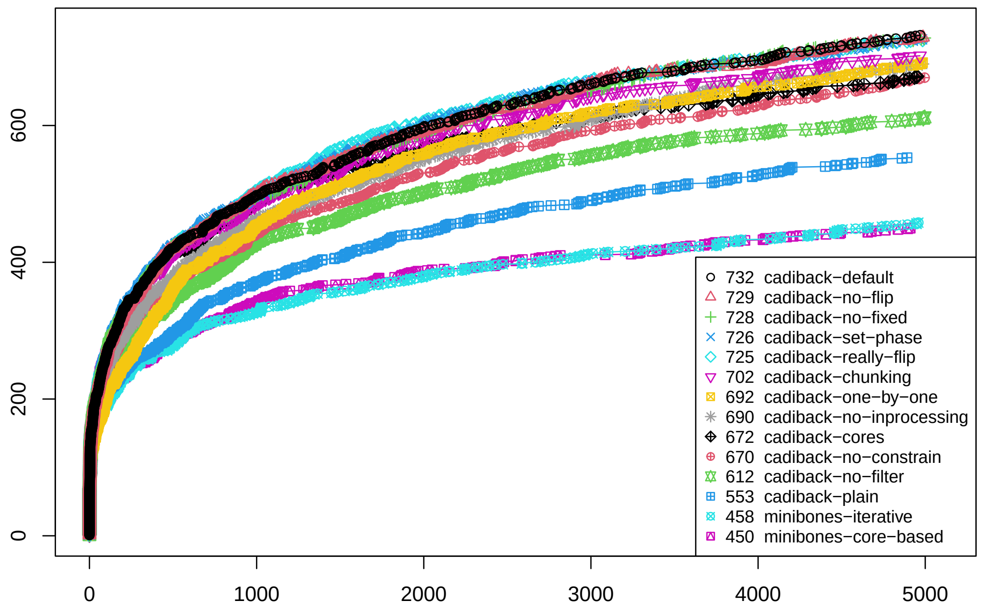

Figure 6.1: Benchmarks solved (vertical) over time in seconds (horizontal) where backbone extraction completed within 5,000 seconds by 12 CADIBACK configurations: default denoting all optimizations enabled except for chunking and cores; no-flip denoting no model rotation; no-fixed representing no checking on candidates for being fixed explicitly; set-phase denoting picking decisions in SAT solver to falsify backbone candidates; really-flip denoting flipping flippable literals eagerly; chunking representing the fine-grained chunk size control ($K = 10$); one-by-one denoting single literal chunks ($K = 1$); no-inprocessing representing no SAT solver internal inprocessing; cores denoting core-based preprocessing; no-constrain meaning only using activation literals instead of using "`constrain`" API; no-filter disables filtering backbone candidates by the disagreement condition; and plain setting $K = 1$ (as one-by-one) and disabling all other optimizations. We also considered 2 MINIBONES configurations: iterative implementing Algorithm 5 in [145]; and core-based implementing Algorithm 7 in [145].

Appendix

This appendix provides additional experimental details. Figure 6.2 shows a reduced version of Figure 6.1 to improve legibility, and Table 6.1 presents further details on the experiment.

We further present a scatter-plot in Figure 6.3 of our best-performing version of CADIBACK versus the best-performing version of MINIBONES.

The left plot in Figure 6.4 emphasizes why the most simplistic backbone algorithm, i.e., assuming the negation of exactly one remaining backbone candidate literal, does not scale, as it just takes way too many SAT calls.

Furthermore, in a number of applications it can be beneficial to get the backbones as soon as they are found, particularly if the backbone search does not terminate. To that end we evaluate MINIBONES and CADIBACK as anytime algorithms and compare the number of backbones they find over time. We modified the default configuration of MINI-

BONES (minibones-core-based, i.e., corresponding to options "`-e -i -c 100`") to print a backbone as soon as it is found and evaluated it against the default version of CADIBACK on the 2022 SAT competition benchmark set. The results are presented on the right in Figure 6.4.

Finally we show in Figure 6.5 the performance of the default version of CADIBACK versus the iterative and core-based versions of MINIBONES on the last three SAT Competitions.

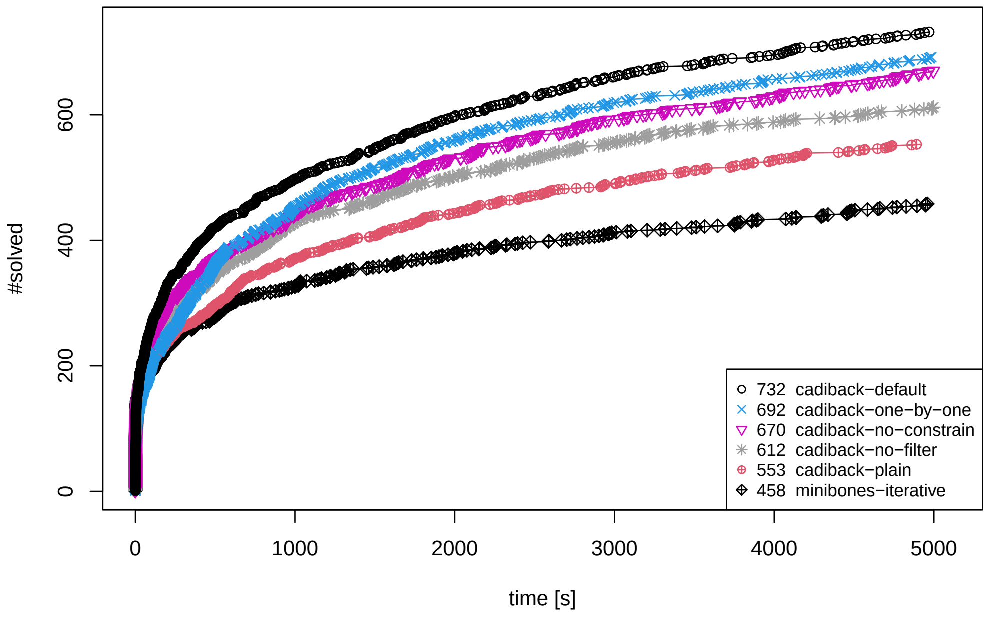

Figure 6.2: Benchmarks solved (vertical) over time in seconds (horizontal) where backbone extraction completed within 5,000 seconds. CADIBACK configurations: default all optimizations enabled except chunking nor cores; no-flip no model rotation (do not use flipping information); no-fixed do not check candidates for being fixed explicitly; chunking fine grained chunk size control ($K = 10$ instead of $K = \infty$); one-by-one single literal chunks ($K = 1$); cores core-based preprocessing; no-constrain activation literals instead of "`constrain`" API; no-filter do not filter backbone candidates by the disagreement condition; plain sets $K = 1$ (as one-by-one) and disables all other optimizations. MINIBONES configurations: iterative Algorithm 5 in [145] ("`-u -c 100 -i`"); core-based Algorithm 7 in [145] ("`-e -c 100 -i`").

	solved	failed	to	mo	time	space	max	best	unique
cadiback-default	732	842	831	11	694 027	110 614	2600	53	1
cadiback-no-flip	729	845	837	8	686 832	103 021	2600	58	0
cadiback-no-fixed	728	846	835	11	682 242	106 129	2600	70	2
cadiback-set-phase	726	848	838	10	657 492	108 737	2565	163	4
cadiback-really-flip	725	849	838	11	640 447	105 963	2600	46	1
cadiback-chunking	702	872	861	11	630 633	93 625	2600	108	0
cadiback-one-by-one	692	882	871	11	715 101	86 152	2600	30	0
cadiback-no-inprocessing	690	884	873	11	688 418	93 947	2628	116	7
cadiback-cores	672	902	891	11	570 724	100 362	2600	78	1
cadiback-no-constrain	670	904	890	14	693 284	93 836	2546	41	0
cadiback-no-filter	612	962	951	11	562 853	72 360	2600	9	0
cadiback-plain	553	1021	1010	11	544 688	58 500	2655	13	0
minibones-iterative	458	1340	1279	61	402 645	110 138	5281	54	17
minibones-core-based	450	1348	1283	65	348 856	72 793	3542	52	2

Table 6.1: More detailed results for the runs plotted in Fig. 6.1 on the large SAT competition 2004 - 2022 benchmark set where: solved instances; failed to solved; to time out of 5,000 seconds hits; mo memory limit of 7 GB hit; time accumulated process time of solved instances (in seconds); space sum of the maximum memory usage over solved instances (in MB); max maximum memory usage on solved instances (in MB); best number of instances with best shortest solving time; unique uniquely solved number of instances. For the description of the configurations see caption of Fig. 6.1.

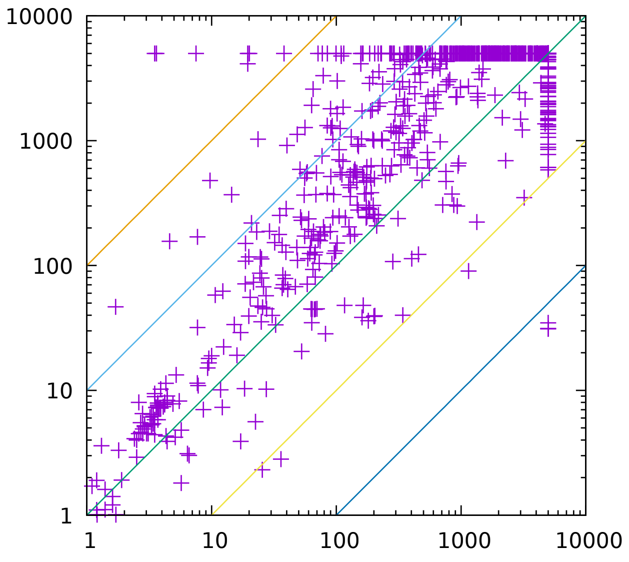

Figure 6.3: Comparing run-time of configuration cadiback-default (horizontal) with configuration minibones-iterative (vertical) solving the large benchmark set from 2004-2022. A cross above the diagonal means CADIBACK is faster than MINIBONES.

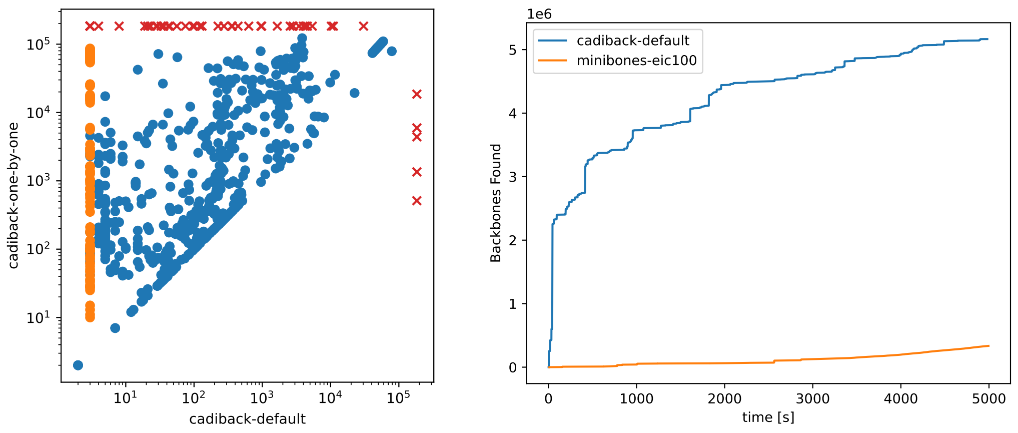

Figure 6.4: The *left plot* compares the one-by-one and the default configuration. Timeouts for one of the configurations are marked in the margin. Highlighted on the left are 133 (out of 1798) benchmarks that have exactly one model (every variable is in the backbone). Using an infinite chunk size (the default), such benchmarks are always solved in 3 SAT calls. The *right plot* compares MINIBONES and CADIBACK in an anytime setting. Shown are the number of backbones found combined across all instances in the SAT competition 2022 benchmark set.

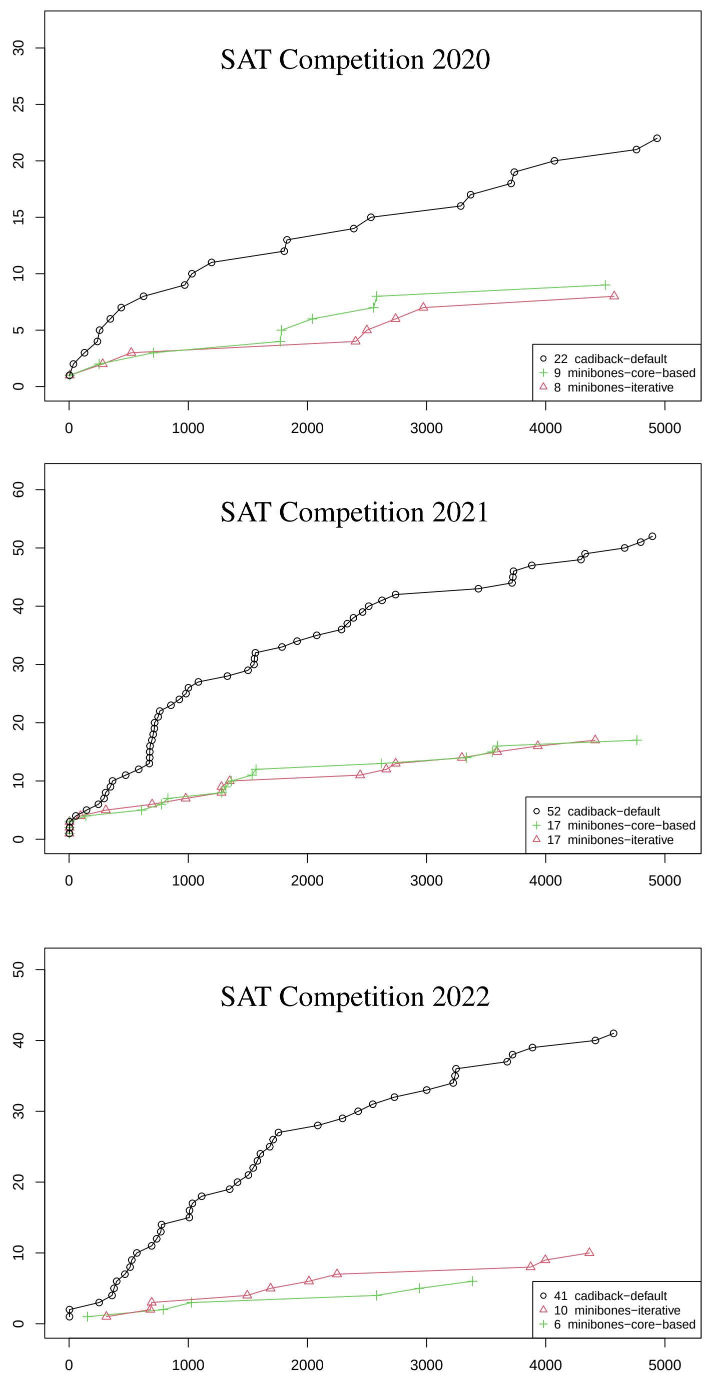

Figure 6.5: Benchmarks solved on satisfiable instances from the last three competitions (2020-2022).

Chapter 7

BIG Backbones

Published In 2023 Formal Methods in Computer-Aided Design (FMCAD)

Authors Nils Froleyks, Emily Yu, and Armin Biere

Changes from Published Version Improved writing in preliminaries. Corrected Typos and adjusted layout.

Authors Contributions The author implemented and evaluated the presented extension in CadiBack, along with the competing algorithms. The KB3 algorithm is based on an idea by A. Biere. The other algorithms, proofs and theoretical results, are contributions by the author.

Acknowledgement This work is supported by the Austrian Science Fund (FWF) under projects W1256-N23 and S11408-N23, and the LIT AI Lab funded by the State of Upper Austria.

Abstract The backbone of a satisfiable formula is the set of literals that hold true in every model. In this paper we introduce Single Unit Resolution Backbone (SURB) which names both a polynomial-time algorithm for backbone extraction and a class of propositional formulas on which it is complete. We show that this class is a superset of the polynomial-time solvable SLUR formulas. The presented algorithm meets a lower bound on time complexity under the strong exponential-time hypothesis. As a second contribution, we present a version that operates on the binary implication graph (BIG) and implement it as a preprocessor in the recently introduced backbone extractor CADIBACK. Experiments on a large number of SAT competition benchmarks show that our implementation results in faster BIG backbone extraction by an order of magnitude. Additionally, incorporating it as a preprocessor enables CADIBACK to identify up to four times as many backbone literals early on.

7.1 Introduction

Backbone extraction has been put forward as an effective technique for a wide variety of applications including chip verification, specifically fault localization [187–189], and interactive configuration [185]. The concept of the backbone, which refers to the set of literals that hold true in all models of a satisfiable formula, was initially studied when investigating the hardness of (random) propositional formulas [181, 205, 206]. Since then, a number of practical applications for backbone extraction have been discovered. One notable example is the improved performance in SAT solving itself [194, 207]. In fact, the proposed algorithm in this paper has been implemented as a cheaper version of failed literal elimination in SAT solvers developed by one of the authors [208]. Other related areas like maximum satisfiability [190–193] have been found to benefit from early knowledge of backbone literals.

In these applications, it is highly advantageous to promptly access as many backbone literals as possible. This can be due to two key reasons: either the backbone computation is bound by a time limit or the identification of a backbone literal triggers additional computations that can be executed in parallel. As a result, our focus shifts to the time taken to identify individual backbone literals rather than the completion of the entire backbone extraction.

The state-of-the-art in backbone extraction has remained unchanged for a long time, until recently CADIBACK [6] was introduced, exhibiting significantly better performance. This was achieved by using the modern SAT solver CADICAL and tightly integrating features currently not found in any other SAT solver. Our contribution presented in this paper is orthogonal to that. Instead of using an exponential approach based on incremental SAT solving, we use a polynomial time algorithm to extract the backbone from a subset of the formula.

In SAT solving, Single Look-ahead Unit Resolution (SLUR) [209], independently discovered as Backtrack-once in [210], defines a class of formulas that are solvable in polynomial time. Similar to that, we define a simple polynomial algorithm called SURB and use it to define a subclass of propositional formulas on which backbone extraction is easy. We formally show that SURB is a strict superset of SLUR. As another novel contribution, we present a practical algorithm that exhibits considerably better performance in our experimental evaluation than SURB. The algorithm finds all backbone literals in the binary implication graph. We implement it as a preprocessor, extending the recently introduced backbone extractor CADIBACK [6]. Results show that our implementation outperforms the previous state-of-the-art by an order of magnitude. Furthermore, our extension enables CADIBACK to identify a subset of all backbone literals within a fraction of the time required to find an initial model.

7.2 Preliminaries

We consider satisfiable SAT formulas in conjunctive normal form (CNF). For a formula $\mathcal{F}$, $\mathcal{V}$ denotes the set of variables, $\mathcal{L}$ the set of literals, n the number of literals, $|\mathcal{F}|$

the number of literal occurrences, $\sigma \subset \mathcal{L}$ commonly denotes an assignment that can also be interpreted as the conjunction of its literals and $\mathcal{F}_{|\sigma} \vdash_1 \ell$ denotes that literal ℓ can be derived by repeated application of the unit propagation rule [211] under σ. The assignment resulting from unit propagation until fixpoint is $\{k \mid \mathcal{F}_{|\sigma} \vdash_1 k\}$. If no conflict arises, the assignment can be extended and the computation repeated until a full assignment is reached. The entire process can be computed in $\mathcal{O}(|\mathcal{F}|)$ [212]. As a convenience, if a conflict is derived we write $\mathcal{F}_\sigma \vdash_1 \lightning$. This notation extends to conflicting assignments $\lightning \in \sigma$. The Binary Implication Graph (BIG) of $\mathcal{F}$ has a node for each literal in $\mathcal{L}$ and two edges $(\neg u, v)$ and $(\neg v, u)$ for each binary clause $\{u, v\}$ [213]. By contraposition, if there is a path from u to v, there is also a path from $\neg v$ to $\neg u$ [214]. Equivalent Literal Substitution (ELS) identifies all cycles in the BIG and replaces them with a single representative. Failed Literal Elimination (FLE) [215] identifies literals ℓ with $\mathcal{F}_{|\ell} \vdash_1 \lightning$ and adds $\neg\ell$ as a unit clause. This is done repeatedly until a fixpoint is reached.

Algorithm 4 (SLUR) [209] may return unsatisfiability, a satisfying assignment, or give up. If it succeeds for any variable ordering, the formula is in the SLUR class [216]. Notable subsets of SLUR include 2-CNF which contain only binary clauses, and Horn-3-CNF that contain length 3 clauses with at most one positive literal.

```
SLUR (CNF $\mathcal{F}$)
1  $\sigma \leftarrow \{k \mid \mathcal{F} \vdash_1 k\}$
2  if $\lightning \in \sigma$ then return UNSAT
3  for $v \in \mathcal{V}$
4     $\sigma^+ \leftarrow \{k \mid \mathcal{F}_{|\sigma \wedge v} \vdash_1 k\}$
5     $\sigma^- \leftarrow \{k \mid \mathcal{F}_{|\sigma \wedge \neg v} \vdash_1 k\}$
6     if $\lightning \in \sigma^+$ and $\lightning \in \sigma^-$ then
7        return GIVE-UP
8     if $\lightning \in \sigma^+$ then $\sigma \leftarrow \sigma^-$
9     else  $\sigma \leftarrow \sigma^+$
10 return SAT, $\sigma$
```

Algorithm 4: Single Look-ahead Unit Resolution. Success depends on the formula and the variable order chosen in line 3.

7.3 Single Unit Resolution Backbone

This section, introduces the algorithm SURB (Single Unit Resolution Backbone) for finding backbone literals and defines a subclass of formulas with the same name.

The algorithm is sound, since the negation of failed literals are backbone literals and only previously identified backbone literals are added to the propagation. In the following we introduce the SURB subclass based on Algorithm 2.

```
SURB (CNF $\mathcal{F}$)
1 $\mathcal{B} \leftarrow \emptyset$
2 for $\ell \in \mathcal{L}$
3     if $\mathcal{F}_{|\mathcal{B}\wedge\ell} \vdash_1 \lightning$ then
4         $\mathcal{B} \leftarrow \{k \mid \mathcal{F}_{|\mathcal{B}\wedge\neg\ell} \vdash_1 k\}$
5 return $\mathcal{B}$
```

Algorithm 5: Single Unit Resolution Backbone identifies a subset of the backbone. The order of literals chosen in line 2 is non-deterministic and can influence which backbone literals are identified.

Definition 7.1. A formula $\mathcal{F}$ is in SURB if the algorithm identifies the entire backbone for any order of literal selection.

The relation of SURB and other classes can be summarized as the following, where FLBE is defined later in Def. 2.

$$\text{2-CNF} \subsetneq \text{SLUR} \subsetneq \text{SURB} \subsetneq \text{FLBE}$$

Similar to SLUR, running Algorithm 2 does not indicate the membership in the class. Deciding if a formula is in SLUR is co-NP-complete [217]. We leave a similar proof for SURB to future work. In practice, this means that without additional knowledge about the formula, it is unknown if the backbone extends beyond the literals identified by SURB. We now formally prove the subset relations from above.

Theorem 7.2. *SLUR ⊂ SURB*

Proof. Assume a satisfiable formula $\mathcal{F}$ has a backbone literal $\neg\ell$ that is not identified by SURB, we show SLUR can fail on $\mathcal{F}$. Let ℓ be the first variable that is decided by SLUR. By the assumption $\mathcal{F}_{|\mathcal{B}\wedge\ell} \not\vdash_1 \lightning$ for some set $\mathcal{B}$ and therefore especially for $\mathcal{B} = \emptyset$. SLUR chooses σ^+ to proceed and will eventually give up since $\neg\ell$ is a backbone literal. □

We use the example below to show that not all formulas in SURB are in SLUR.

Example 7.3. Consider the formula $\mathcal{F} = (\neg a \vee b \vee \neg c \vee d) \wedge (\neg a \vee b \vee \neg c \vee \neg d) \wedge (\neg a \vee b \vee c \vee d) \wedge (\neg a \vee b \vee c \vee \neg d)$.

SLUR fails for the variable order $[a, b, c, d]$. However, $\mathcal{F}$ has neither failed nor backbone literals.

Definition 7.4. Failed Literal Backbone Equivalent (FLBE) is the class of formulas on which the negation of every backbone literal is a failed literal.

This class defines the upper bound on which SURB is complete, if it had an oracle to determine the optimal ordering of literals to propagate. Without the correct ordering, SURB would need to be executed up to n times to identify the entire backbone of a formula in FLBE. The following example illustrates this.

Example 7.5. Consider the formula
$\mathcal{F} = (\neg a \vee \neg b) \wedge (\neg a \vee b) \wedge (a \vee \neg c \vee \neg d) \wedge (a \vee \neg c \vee d)$.
If c is propagated before a, only $\neg a$ will be found by SURB. However, both a and c are failed literals and there are no further backbone literals, thus $\mathcal{F}$ is in FLBE.

Now we proceed to discuss the time complexity of SURB. It performs up to n propagations and therefore has a worst-case complexity of $\mathcal{O}(n \cdot |\mathcal{F}|)$. Järvisalo and Korhonen [218] suggest that any algorithm to find even a single backbone literal in a Horn-3-CNF has worst-case complexity of $\mathcal{O}(n \cdot |\mathcal{F}|)$ under the strong exponential time hypothesis [219]. Since SURB subsumes the problem and is complete on a superset of Horn-3-CNF, it is unlikely that we can achieve a better worst-case complexity than what this simple algorithm offers.

The same asymptotic time complexity is also shared by SLUR [209]. However, while SLUR continuously extends an assignment and uses it for future propagations, SURB only saves backbone literals. As in the end both algorithms propagate each literal at least once, keeping more literals assigned can lead to a faster overall runtime. We exploit this idea in the design of Algorithm 6 in the next section.

7.4 BIG Backbones

As the algorithm presented in the previous section is generally not guaranteed to identify the entire backbone of a formula, it can only serve as part of the backbone search. Applying SURB to the entire formula would be too slow, even with the highly optimized implementations of unit propagation in modern SAT solvers. It is also not possible to efficiently identify the subset of a formula that is in SLUR [217]. We therefore focus on the binary clauses where propagation can be implemented more efficiently and SURB is complete. The following proposition justifies focusing on a subset.

Proposition 7.6. *The backbone found on a subset of a satisfiable formula $\mathcal{F}$ is a subset of the backbone of $\mathcal{F}$.*

In Algorithm 6, we present KB3, a version of SURB, which is only valid for 2-CNFs and avoids re-propagation by keeping an assignment between propagations.

The example in Figure 7.1 illustrates why we can keep literals assigned without encountering spurious conflicts. Specifically, running the KB3 algorithm for this formula, when c is picked as the first candidate (line 4), all candidates with a path to $\neg c$ are blocked until the assignment is reset in line 3.

Theorem 7.7. *Algorithm 6 is sound and complete on 2-CNF.*

Proof. Since 2-CNF is a subset of SURB we can rely on the completeness of Algorithm 5 for any variable ordering. We can therefore assume the set $\mathcal{B}$ to be empty for every candidate ℓ in line 3. Every literal is either eliminated in line 7 or eventually propagated. Note that propagation under an assignment is only more likely to derive a conflict. Any eliminated literal has been assigned by a previous propagation that did not lead to a conflict.

```
KB3 (2-CNF $\mathcal{F}$)
1  $\mathcal{B} \leftarrow \emptyset, \quad \Lambda \leftarrow \mathcal{L}$
2  while $\Lambda \neq \emptyset$
3      $\sigma \leftarrow \mathcal{B}, \quad \Delta \leftarrow \emptyset$
4      for $\ell \in \Lambda$ // next candidate
5          if $\neg\ell \in \sigma$ then continue
6          $\Delta \leftarrow \Delta \cup \{\ell\}$
7          if $\ell \in \sigma$ then continue
8          $\sigma' \leftarrow \{k \mid \mathcal{F}_{|\sigma\wedge\ell} \vdash_1 k\}$
9          if $\lightning \in \sigma'$ then
10             $\mathcal{B} \leftarrow \mathcal{B} \cup \{k \mid \mathcal{F}_{|\neg\ell} \vdash_1 k\}$
11             $\Delta \leftarrow \Delta \cup \mathcal{B} \cup \neg\mathcal{B}$
12             $\sigma \leftarrow \sigma \cup \mathcal{B}$
13         else $\sigma \leftarrow \sigma'$
14     $\Lambda \leftarrow \Lambda \setminus \Delta$
15 return $\mathcal{B}$
```

Algorithm 6: Keep assignment BIG Backbone (KB3) is only defined on 2-CNFs, for which it is complete regardless of the literal selection order (in line 4).

$$\begin{tikzcd}
\neg a \arrow[r] & \neg b \arrow[r] \arrow[dr] & \neg c \arrow[r] \arrow[dr] & \neg d & \\
& d \arrow[r] & c \arrow[r] & b \arrow[r] & a
\end{tikzcd}$$

Figure 7.1: BIG of $(a \vee \neg b) \wedge (b \vee \neg c) \wedge (b \vee c) \wedge (c \vee \neg d)$.

To show soundness, consider a conflict derived in line 8. Let c and $\neg c$ be any pair of conflicting literals and ℓ the current candidate. We show neither c nor $\neg c$ are in σ and therefore $\mathcal{F}_{|\ell} \vdash_1 \lightning$ which implies that $\neg\ell$ is a backbone literal. It is impossible for c and $\neg c$ to both be in σ since the conflict would have prevented the assignment from being updated in line 13. Without loss of generality assume c to be in σ and the propagation of ℓ to imply $\neg c$. Since ℓ implies $\neg c$, there is a path from ℓ to $\neg c$ in the BIG and by contraposition there is also a path from c to $\neg\ell$. The set σ is the result of propagation therefore every literal implied by c is included. But if $\neg\ell \in \sigma$ the current candidate would have been skipped in line 5. $\square$

By this proof, ℓ is a failed literal in the original formula. Therefore a resolution proof for $\neg\ell$ being in the backbone can be found by resolving the clauses corresponding to the paths from ℓ to c and ℓ to $\neg c$ in the BIG.

The example below shows that Algorithm 6 does not extend to Horn-3-CNF.

Example 7.8. Consider the formula $(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$. Both $\neg a, b, c$ and $a, \neg b, \neg c$ satisfy the formula, the backbone is therefore empty. However, if the candidates are picked in the order $[a, b, \ldots]$ literal $\neg b$ is identified as part of the backbone.

7.5 Related Techniques

We now discuss some previous work on extracting backbones from BIGs [214], as well as other techniques used in failed literal extraction. We refer to Figure 7.1 for an illustration of the following discussions. The algorithm Van Gelder describes in [214] is essentially equivalent to SURB with a depth-first search order, instead of the usual breadth-first propagation. Whenever the BFS propagation of a literal $\neg a$ causes the assignment of conflicting literals c and $\neg c$, the BIG does not only contain a path from $\neg a$ to c and $\neg a$ to $\neg c$ but by contraposition also a path from c to a. Thus, with a DFS order the first conflicting literal b, is always in the backbone. Furthermore, b is the highest such literal in the search tree, so propagating it will identify all other backbones that can be found for this conflict. To emulate this desirable property with BFS, we can explicitly store the search tree and identify the first `UIP`[220] after a conflict occurs.

`Stamping` [60, 221] prevents a literal to be considered as a candidate, if it has been propagated since the last backbone literal was identified. However, such a literal must still be re-propagated if it is encountered during another propagation, as the previous example demonstrates for the candidate order $[d, \neg a, \ldots]$. KB3 subsumes this technique, since any candidate that is not propagated due to stamping would still be assigned and added to Δ in line 6, at the first time it is encountered. Moreover, while stamping is reset when a conflict is encountered, KB3 still maintains part of the assignment.

`Roots` [70, 214, 222] only propagates a candidate if it has no predecessor in the BIG. Removing the negation of an identified backbone literal can add new roots. This optimization is part of Van Gelder's algorithm and also used in failed literal elimination. To maintain completeness, it is necessary to run ELS until fixpoint, if only the BIG is considered one round is sufficient. Since a root cannot be implied by another candidate, this technique also subsumes `Stamping`. Note that combining this technique with KB3 increases the size of the unkept assignment when a conflict is encountered and can therefore also have negative effects.

We present two scalable examples of 2-CNF formulas. Figure 7.2 is lifted from [214]. They used the example to show that their algorithm expands $\mathcal{O}(n^3)$ edges and is therefore not more efficient than computing the two-closure. In contrast, our algorithm expands each edge exactly once and is thus in $\mathcal{O}(n^2)$. We can therefore achieve a speedup of n times and reach the lower bound complexity of performing a single propagation. However, as the example in Figure 7.3 shows, the worst case complexity has not changed. Each of the $\mathcal{O}(n)$ roots in group R expands the $\mathcal{O}(n^2)$ edges in P and the at-most-one constraint prevents any of the propagations from being reused.

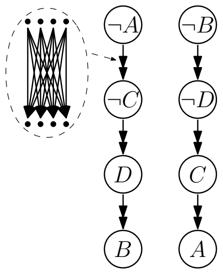

Figure 7.2: The positive literals are split into four equal groups. The double-arrow denotes that each literal of one group implies all literals of the other [214].

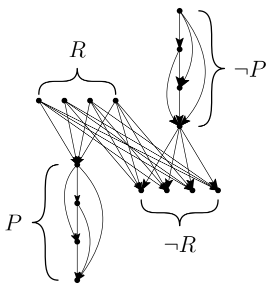

Figure 7.3: The literals in R are connected to $\neg R$ with an at-most-one constraint, meaning that each literal has an edge to the negation of every other literal. They all connect to the highest literal in P. Literals in P have an edge to every lower literal.

7.6 Implementation and Evaluation

We implement the new algorithm KB3 [32] and the base version SURB with various optimizations as preprocessors for CaDiBack [6]. For each configuration we tested both DFS and BFS for propagation. The binary clauses are extracted after some basic preprocessing has been performed by CaDiCaL and stored as an adjacency array. All backbone literals in the BIG are then extracted and added as unit clauses before the first call to a SAT solver. To increase trust, we checked that all configurations identify the same backbone on close to a billion randomly generated 2-CNF. We use a cluster with 20 nodes each running two AMD EPYC 7313 at 3.7Ghz under Ubuntu 22.04 LTS. Memory is limited to 15GB per instance.

For benchmarking, we collected formulas from SAT competitions 2004-2022, and removed duplicates to obtain a large and representative set. We ran Kissat 3.0.0 [144] for 5,000 seconds to identify satisfiable benchmarks. This left us with 1798 benchmarks (available at https://cca.informatik.uni-freiburg.de/sc04to22sat.zip (6 GB) and [31]).

Table 7.4 presents the comparison of the different configurations. Even though the source code from [214] is not available, the configuration of SURB with DFS and ELS+Roots in our implementation is equivalent to what they describe. The results show that the new algorithm clearly outperforms the configuration of [214], being more

	SURB		KB3	
	BFS	DFS	BFS	DFS
Base	21136.11	21287.25	647.53	728.46
ELS	20523.81	20756.81	**640.43**	733.47
ELS+Roots	18164.47	18756.10	643.57	721.09
stamp	19205.73	19636.01		
ELS+Stamping	18947.99	19392.12		
ELS+Roots+UIP			822.49	

Figure 7.4: The time in seconds to run backbone extraction on the BIG until completion accumulated over all satisfiable benchmarks from the last 19 SAT competitions (2004-2022). The time to run ELS on the entire benchmark set is 50.59 seconds and included for the algorithms which use it.

than 29 times faster. Three benchmarks are particularly hard for SURB. Only the BFS configurations without stamping solve them within the time limit of 5000 seconds, whereas KB3 takes less than a second to solve them. Furthermore, the additional optimizations work well for the base version, however, as expected KB3 does not seem to benefit from them as much.

As argued before, KB3 subsumes stamping and the combination is therefore not presented. Similarly, the UIP technique is not necessary when a depth first order is used for propagation and has not been implemented for SURB.

In the second part of the evaluation we investigate how the best configuration of KB3 (BFS and ELS) performs as a preprocessor for the complete backbone extractor CaDiBack. We log the time of identifying a backbone literal for the 533 benchmarks from the past three SAT competitions (2020-2022) and present their accumulation over time in Figure 7.5. Even though we limit the run time to 1000 seconds, still more than 10 Million backbone literals are identified. The version with KB3 holds the biggest absolute advantage at around 210 seconds, where it identified 5.5 Million backbone literals, 4.5 times as many as the base version has found at that point.

7.7 Conclusion

In this paper we proposed a new algorithm for backbone extraction from the binary implication graph of a formula. The new algorithm exhibits a significant performance advantage over the previous state-of-the-art approach. Furthermore, we have integrated our algorithm into the backbone extractor CaDiBack as a preprocessor, yielding remarkable improvements, particularly in the early identification of backbone literals.

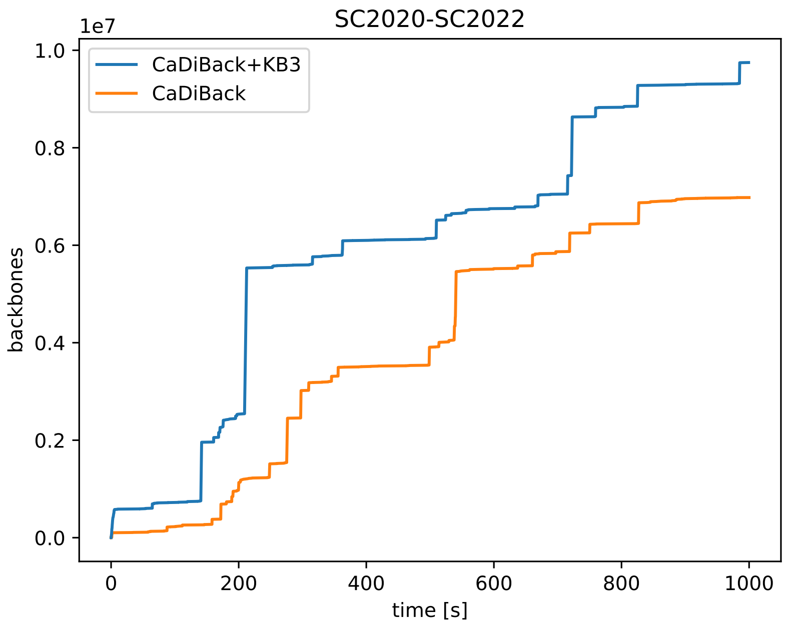

Figure 7.5: Presented is the number of backbone literals identified over time. We compare default CADIBACK to a version with added preprocessing performed by KB3.

Chapter 8

Ternary Simulation as Abstract Interpretation

Published In MBMV 2024; 27. Workshop

Authors Nils Froleyks, Emily Yu, and Armin Biere

Changes from Published Version Edited the title and removed the explicitly stated open questions to eliminate the "Work in Progress" aspect. Added a concluding note on the effectiveness of narrowing. Fixed typos.

Authors Contributions The author provided the formalization, devised and implemented the widening, backbone-based narrowing, and subsumption-based termination.

Acknowledgement This work is supported by the Austrian Science Fund (FWF) under the project W1255-N23, the LIT AI Lab funded by the State of Upper Austria, and Academy of Finland under the project 336092.

Abstract We introduce a formalization of ternary simulation as abstract interpretation along with a widening operator to speed up convergence. With the same goal, we present a subsumption algorithm that can determine termination earlier than the usual approach using hash sets. Additionally, we introduce a narrowing operator that utilizes recent advances in backbone extraction, allowing to increase the overapproximation precision in simulation at any time. The experiments evaluate the presented techniques in the context of hardware model checking.

8.1 Introduction

Optimizing reachability analysis and model checking is an important topic in formal verification of hardware designs. Ternary simulation [223] is a powerful method utilized in various preprocessing techniques as well as tightly integrated in standard model checking approaches like PDR [224]. For example, phase abstraction [223] leverages ternary simulation for identifying a group of oscillating signals used to simplify the original hardware designs.

By using a three-valued logic $\{1, 0, X\}$, at initialization the inputs are assigned unknown values (X) while latch variables are assigned according to their reset definitions; the successor states are then computed based on the three-valued semantics. Differently from using conventional symbolic execution, as a result an over-approximation of reachable states is obtained at termination. This also allowed us in recent work on model checking certificates to provide an alternative notion of cube semantics [13].

Even though ternary simulation enables fast computation in practice even on industrial designs [225], the key challenge, is the exponential time for convergence in the worst case, due to the PSPACE complete nature of symbolic reachability analysis, also known as the "state explosion" problem. Furthermore, while ternary simulation has been implemented in numerous state-of-the-art tools, there are still significant gaps between its precise theory and practical applications.

On another matter, abstract interpretation [226] helps to obtain sound and precise overapproximations of the state space, and has been commonly used in the static analysis of software systems [227]. It relies on a logical approximation relation between concrete models and abstract predicates to produce a sound fixpoint approximation.

Related to ternary simulation is Symbolic Trajectory Evaluation (STE) [228], which can be characterized as a combination of symbolic execution and ternary simulation, but with the ternary value functions encoded as BDDs.

Previous work [229] has shown, from a theoretical point of view, the Galois connection between a ternary model and Boolean model as a form of abstract interpretation. We focus on practical applications of abstract interpretation and further utilize narrowing and widening operators to refine lossy unknown values and ensure early termination.

In this paper, we make an attempt to bridge the gap by formalizing ternary simulation as a form of abstract interpretation, which is more succinct than the formalism presented in [229], and at the same time closer to practical implementations of simulation.

This enables us to leverage the framework of abstract interpretation with decades of extensive research to enhance bit-level hardware verification. We begin by defining an abstract domain and transformation functions for ternary simulation. Furthermore, we introduce a widening operator in the strict sense of abstract interpretation, guaranteeing early termination to avoid exponential computation. We also present a weaker widening operator, which might be more suitable for practical applications and an algorithm for efficiently determining convergence. More importantly, we further enhance the technique by making use of backbone extraction [180] in defining the narrowing operator. Lastly, we also demonstrate the effectiveness of our method in the experimental evaluation.

8.2 Circuit

In the rest of the paper, we assume standard semantics for Boolean operators [230] and use the notations from abstract interpretation theory [231].

Since ternary simulation relies on the structure of a Boolean circuit, our definition is more detailed than a simple transition relation and defines transition functions in the form of an and-inverter graph (AIG).

Definition 8.1 Circuit**.** A Boolean circuit C is represented by the tuple (I, L, A, R, F, D) where $I = \{i_1, \ldots, i_{\#I}\}, L = \{l_{\#I+1}, \ldots l_{\#I+\#L}\}, A = \{l_{\#I+\#L+1}, \ldots, l_{\#I+\#L+\#A}\}$ are *inputs*, *latches* and (AND) *gates* respectively and are ordered continuously. Let $\overline{S} = S \cup \{\neg\ell \mid \ell \in S\}$ for any such set S denote the set of literals over S and `VAR` the inverse operation (determining variables from literals). With that, the set of *all* literals is $\Lambda = \overline{I} \cup \overline{L} \cup \overline{A}$.

The (total) functions $R : L \to \{0, 1\}$ and $F : L \to \Lambda$ define the initial state of the circuit and the transition behavior of the latches respectively. The (total) function $D : A \to \Lambda \times \Lambda$ gives the definition of each AND gate and has to be stratified, i.e.: $\forall a_i \in A : D(a_i) = (a_j, a_k) \Rightarrow j < i \wedge k < i$.

Definition 8.2 Cubes and States**.** For a set of literals S we further define the following sets:

- $c \in \mathbb{P}(S)$ is called a *cube* with $\mathbb{P}(S)$ the power set of S,
- $\mathbb{P}_1(S) = \{c \in \mathbb{P}(S) \mid \ell \in c \Rightarrow \neg\ell \notin c\}$ is the set of *consistent* cubes, and
- $\mathbb{P}_2(S) = \{s \in \mathbb{P}_1(S) \mid |s| = |S|\}$ are the *complete* cubes.

Additionally, we define $\nu : \mathbb{P}_1(\Lambda) \times \Lambda \to \{0, 1, X\}$ to get the value of a literal in a consistent cube and further use $\tau : \Lambda \times \{0, 1, X\} \to \Lambda \cup \{\epsilon\}$ to change the sign of a literal:

$$\nu(c, \ell) = \begin{cases} 1, & \text{if } \ \ell \in c \\ 0, & \text{if } \neg\ell \in c \\ X, & \text{otherwise} \end{cases} \qquad \tau(\ell, b) = \begin{cases} \ell, & \text{if } b = 1 \\ \neg\ell, & \text{if } b = 0 \\ \epsilon, & \text{otherwise} \end{cases}$$

These are the only functions that explicitly deal with three-valued semantics $\{0, 1, X\}$. Otherwise our formalism uses cubes instead of full assignments mapping to $\{0, 1, X\}$. To further simplify the exposition we use ϵ as the neutral element of set-addition, i.e., any set containing ϵ is equivalent to the same set without it.

The stratification condition in Def. 8.1 ensure that the directed graph induced by D is a acyclic, i.e., a *directed acyclic graph* (DAG), with inputs and latches as leaves. This makes the transition function of a circuit well-defined:

Definition 8.3 Transition**.** For a circuit $C = (I, L, A, R, F, D)$ we define the following functions:

- $ext_I : \mathbb{P}_2(L) \to \mathbb{P}(\mathbb{P}_2(I \cup L))$,
  $ext_I(s) = \{s' \in \mathbb{P}_2(I \cup L) \mid s' \Rightarrow s\}$
  that expands a state over latches with all possible inputs.

- $ext_A : \mathbb{P}_2(I \cup L) \to \mathbb{P}_2(\Lambda)$ is defined as fixpoint of
$\varphi(s) = \{\tau(a, \texttt{AND}(\nu(s,l), \nu(s,r))) \mid (a,(l,r)) \in D\}$
and can be computed in a single pass over the gates due to the stratification assumption. Since s is a complete state and therefore $\nu(s,\cdot)$ maps to $\{0,1\}$, AND computes the standard Boolean function ("$\wedge$").
- $nxt_L(s) = \{\tau(\ell, \nu(s,n)) \mid (\ell, n) \in F\}$
extracts the successor state from next-state function of a latch.
- $f^L : \mathbb{P}_2(L) \to \mathbb{P}(\mathbb{P}_2(L))$, with
$f^L(s) = \{nxt_L(ext_A(s')) \mid s' \in ext_I(s)\}$,
maps a state to the set of its successors in C.

8.3 Ternary Simulation as Abstract Interpretation

In this section we define the abstract transformation from a concrete circuit to a ternary circuit. We will consider two lattices $(\Sigma, \subseteq)$, with $\Sigma = \mathbb{P}(\mathbb{P}_2(L))$ for the concrete semantics of the circuit and $(\Omega, \Rightarrow)$, with $\Omega = \mathbb{P}(L)$ for abstract semantics utilizing ternary simulation.

We prefer to use the set of cubes Ω instead of three-valued states: $\{0,1,X\}^{|L|} \cup \bot$. Here we use $\bot$ to denote "*no-state*" (inconsistent cube in $\mathbb{P}(L)$), not to be confused with the ternary state $(X, X, \dots)$ representing all states equivalent to the empty cube in our abstract domain.

Theorem 8.4. *$(\Sigma, \subseteq)$ and $(\Omega, \Rightarrow)$ are complete lattices.*

We further define two functions to translate between them:

Definition 8.5. The *abstraction function* $\alpha : \Sigma \to \Omega$ is defined as $\alpha(\sigma) = \bigcap_{s \in \sigma} s$.

Definition 8.6. The *concretization function* $\gamma : \Omega \to \Sigma$ is defined as $\gamma(\omega) = \{s \mid s \in \mathbb{P}_2(L) \wedge s \Rightarrow \omega\}$.

Theorem 8.7. *The tuple $(\gamma, \Sigma, \Omega, \alpha)$ is a Galois connection.*

Proof. For sets of states $\sigma \in \Sigma$ and cube $\omega \in \Omega$, we have:

$$\begin{array}{lll}
& \alpha(\sigma) \Rightarrow \omega & \text{by Def. 8.5} \\
\Leftrightarrow & (\bigcap_{s \in \sigma} s) \Rightarrow \omega & \\
\Leftrightarrow & \forall s \in \sigma : s \Rightarrow \omega & \text{with } \sigma \subseteq \mathbb{P}_2(L) \\
\Leftrightarrow & \sigma \subseteq \{s \mid s \in \mathbb{P}_2(L) \wedge s \Rightarrow \omega\} & \text{by Def. 8.6} \\
\Leftrightarrow & \sigma \subseteq \gamma(\omega) &
\end{array}$$ □

For the concrete transition function, we define the transition of a set of states as the union of their successors.

Definition 8.8. The *concrete transition* function $f : \Sigma \to \Sigma$ is defined as $f(\sigma) = \bigcup_{s \in \sigma} f^L(s)$.

On the abstract side we finally formally define ternary simulation as the abstract transition function that allows us to transition in the abstract domain.

Definition 8.9. We define $f^\# : \Omega \to \Omega$ as the *abstract transition function* with $f^\#(\omega) = nxt_L(ext^{\mathtt{X}}_A(\omega))$, where $ext^{\mathtt{X}}_A : \mathbb{P}_1(I \cup L) \to \mathbb{P}_1(\Lambda)$ follows the definition of ext_A except AND is replaced by $\mathtt{AND}^{\mathtt{X}}$ defined by the table below:

l	r	$\mathtt{AND}^{\mathtt{X}}(l, r)$
0	-	0
-	0	0
1	1	1
1	X	X
X	1	X

where '-' denotes either 0 or 1.

We have to show that $f^\#$ is indeed a valid abstraction of f.

Theorem 8.10. $(f \circ \gamma)(\omega) \subseteq (\gamma \circ f^\#)(\omega)$

For that we will first define the depth of a gate.

Definition 8.11 Depth**.** The *depth* of a gate $a \in A$ is defined as the length of the longest path from a to a leave in the DAG induced by D. The depth of an input I or latch L is 0.

We state a connection between the transition functions.

Lemma 8.12. *For $\omega \in \mathbb{P}_1(L)$, $s \in \gamma(\omega)$, $a \in I \cup L \cup A$, if $\nu(ext^{\mathtt{X}}_A(\omega), a) \in \{0, 1\}$ then $\nu(ext_A(s), a) = \nu(ext^{\mathtt{X}}_A(\omega), a)$.*

Proof. The proof proceeds by induction over the depth n of a. For $n = 0$, a is either an input or a latch literal, where only the latch might be in ω in which case it will also be in both extensions. Now for any gate a at depth $n + 1$ let $(l, r) = D(a)$, both l and r have depth no greater than n. Further, in case $c = \mathtt{X}$ the claim holds trivially. Otherwise, neither l nor r are X, thus AND equals $\mathtt{AND}^{\mathtt{X}}$ or exactly one of them is X and the other one is 0. In the latter case, both extensions in $\gamma(\omega)$ result in 0. $\square$

We continue to the proof of the Theorem 8.10.

Proof of Theorem 8.10. Suppose there is a $\sigma \in (f \circ \gamma)(\omega)$ that differs from all states in $(\gamma \circ f^\#)(\omega)$ in at least one literal. Let ω be as stated in the theorem, $\sigma = \gamma(\omega)$ and σ' a state in $(\gamma \circ f^\#)(\omega)$ that differs from σ in the fewest number of literals, one of them being ℓ. We consider two cases: (1) $\nu(f^\#(\omega), \ell) = \mathtt{X}$: By definition of γ the set $\gamma(f^\#(\omega))$ also contains a state that is the same as σ' but matches $\sigma \in \ell$, contradicting our assumption. (2) $\nu(f^\#(\omega), \ell) = c$, with $c \in \{0, 1\}$: Let $a = F(\ell)$, then $\nu(ext^{\mathtt{X}}_A, a) = c$. By Lemma 8.12 and Def 8.8 σ matches σ' on ℓ again contradicting our assumption. $\square$

8.4 Widening

Even though ternary simulation can be implemented very efficiently, it is exponential in the size of the circuit. In fact, this exponential behavior is easily exposed by adding a 64-bit counter that is independent of the rest of the design.

The widening operators, as introduced in [232], promise to alleviate that problem by guaranteeing a faster convergence. However the demands on the properties of such a widening operator are quite strong. We will introduce ∇ that fulfills both the *covering* and *termination* property [233] and additionally the more conservative operator $\overline{\nabla}$ that does not meet these strict criteria, but exhibit superior performance in our application. A similar operation has been introduced as *X-saturation* in [225].

Definition 8.13. $\nabla : \mathbb{P}(L) \times \mathbb{P}(L) \to \mathbb{P}(L),\ a \nabla b = a \cap b.$

Theorem 8.14. ∇ *is:*

1. covering*:* $\forall a, b \in \mathbb{P}(L) : a \Rightarrow a \nabla b$*, and* $b \Rightarrow a \nabla b$
2. terminating*: For an ascending chain* $\{a_i\}_{i \geq 0}$*, the chain* $b_0 = a_0, b_{i+1} = b_i \nabla a_{i+1}$ *stabilizes after a finite number of terms.*

Definition 8.15. $\overline{\nabla} : \mathbb{P}(L) \times \mathbb{P}(L) \to \mathbb{P}(L)$ with
$a \overline{\nabla} b = b \setminus \{\ell\}$ and where neither $\ell \in b, \neg\ell \in a$ nor $\mathtt{VAR}(\ell)$
have been removed by widening before.

8.5 Narrowing

Ternary simulation can produce a high number of spurious traces, which is even more true if widening is used. Narrowing operators [233] increase the precision of the simulation at any point, thus removing a set of spurious traces while still maintaining a valid over-approximation.

Our narrowing operator for ternary simulation Δ relies on the backbone of the transition between two cubes. The backbone of a satisfiable formula, is the set of literals that hold true in all assignments. It is only applicable to two cubes a, b if $(f^{\,L})^n(a) \Rightarrow b$, were n is the number of function applications. For simplicity we will only define it for a single step of the transition function.

Definition 8.16. $\Delta : \mathbb{P}(L) \times \mathbb{P}(L) \to \mathbb{P}(L)$ with $a \,\Delta\, b = F^{-1}(B(a \wedge D \wedge F(b)))$, and where

- $F(b) = \{\tau(F(\ell), \nu(b, \ell)) \mid \ell \in \mathtt{VAR}(b)\}$ denotes the "primed" version of a cube b,
- $F^{-1}(c) = \{\tau(\ell, \nu(c, n)) \mid (\ell, n) \in F\}$ the inverse and
- $B(\Phi)$ denotes the backbone of a Boolean formula Φ.

8.6 Termination via Subsumption

In both, collection semantics of abstract interpretation [231] and ternary simulation [225], termination is defined as a subsumption check, i.e., the simulation terminates if a cube implies a previously encountered cube. At that point the encountered cubes represent an overapproximation of all reachable states. Such a check can be implemented using BDDs, however as the authors of [225] state: “In practice, the performance of such approach is prohibitive”. They instead use a hash table and only terminate, when an exact match to a previously encountered cube is found.

We utilize a different algorithm used for *forward subsumption* detection in SAT solving [123, 230]. The algorithm relies on a one-watch data structure, i.e., each cube appears in the watch-list of one of its literals.

```
subsumed (cube c)
1  mark all literals in c
2  for literal $\ell$ in c do
3      for cube c' in watch[$\ell$] do
4          $\ell' \leftarrow$ unmarked literal in c'
5          if $\ell' =$ invalid then // No such literal
6              lassos.add(pred(c), c')
7              if $|c'| = |c|$ then // Exact match
8                  return lassos
9          else watch[$\ell'$].add(c')
10     watch[$\ell$].clear()
11 unmark all literals
```

Algorithm 7: Subsumption check. Identifies all previous cubes that imply c, and thereby induce a *lasso* in the state space. Requires one literal of each cube to be *watched*.

Whenever the algorithm reaches line 6, a sound overapproximation is found. However, for some applications it can be beneficial to consider more than one *cube lasso*. For example in phase abstraction [223] the length of both the stem and the loop of the lasso should be divisible by some small number.

8.7 Evaluation

Our preliminary implementation does not cover the entirety of Sect. 8.4 and 8.5. The simulation itself is fairly efficient, calculating all the gates in a single linear pass, using a few basic bit operations. However, we do not reorder gates to allow for more efficient packing / random access of state bits or parallelization using SIMD / threads.

Termination is implemented both with hashing and forward-subsumption for comparison. We also provide an implementation of the widening operator $\overline{\nabla}$. Ours differs from

	#Latches	base	widening	termination
bob12m04	43950	199 I 0.06	199 I 0.07	153 I **0.05**
bob12m15	448	133 I 1.62	133 I 1.60	12 I **0.40**
bobsmnut1	644	107 I 0.21	107 I 0.19	106 I **0.18**
shift1add	27	20 I 1.20	20 I 1.25	1 I **0.01**
6s376r	4708	145 I 766.34	116 I 16.30	116 I **4.20**
6s47	815	to	8 I 35.63	6 I **0.30**
6s100	97598	to	36 I **37.53**	36 I 41.48
6s107	1568	to	746 I 16.93	746 I **1.18**
6s149	12781	to	4 I 47.81	4 I **12.88**
6s202b41	68881	to	8574 I 45.71	8574 I **28.35**
6s204b16	28986	to	4034 I 19.33	4034 I **13.51**
6s205b20	68842	to	8727 I 46.56	8727 I **28.08**
6s355rb$_{8740}$	15091	to	221 I 37.07	221 I **15.51**
6s400rb$_{7819}$	14665	to	221 I 36.96	221 I **11.00**
6s342rb$_{122}$	56838	to	to	1894 I **354.60**
cucnt128	128	to	to	0 I **1.82**
cucnt32	32	to	to	0 I **0.01**

Table 8.1: We evaluated three versions: **base** ternary simulation with hashing and no widening, a configuration using **widening** ($\overline{\nabla}$) if the cube has not reduced in size in a few thousand iterations, and one that employs *both* widening and the early **termination** using the forward subsumption algorithm. Presented is the runtime and the number of *transients* that could be found for each circuit. Our benchmarkset included all 20 815 circuits from the HWMCC (2007-2020)[20]. The table lists all instances where either transients were lost to the optimization or any of the configurations timed out (to).

the *X-saturation* described in [225] in that we do not eliminate all non-fixed latches, but only pick a single one, which has not been affected by widening before. As ∇ removes too many literals, it is not evaluated. We do not yet have an implementation for narrowing.

All experiments were conducted on our cluster with Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz, with a time limit of 2 hours. As an example application of ternary simulation and to gauge its precision, we extract *transients*. Transients are latches that assume a constant value after a finite number of steps (constant in the loop of any cube lasso). The results are shown in Table 8.1.

Considering the high number of benchmarks, both widening and earlier termination had very little impact on the number of identified transient. However, they did help with a number of related benchmarks that originally exceeded the two-hour time limit. Note that the final configuration using both techniques solved all instances.

Lastly, we ran narrowing at every step until completion. Completely disregarding the runtime, this lead to an increase of 42% in identified transients.

Chapter 9

Certifying Phase Abstraction

Published In International Joint Conference on Automated Reasoning (IJCAR) 2024

Authors Nils Froleyks, Emily Yu, Armin Biere, and Keijo Heljanko

Changes from Published Version Corrected typos and adjusted layout.

Authors Contributions The original certificate generation for phase abstraction was developed by E. Yu. The author generalized the phase abstraction preprocessing technique along with the certificate construction, and implemented both. During fuzzing, a bug in the theory was uncovered and is fixed in this version. The author further added an aspect to the proof of correctness for the certificate format, which was missing in earlier versions.

Abstract Certification helps to increase trust in formal verification of safety-critical systems which require assurance on their correctness. In hardware model checking, a widely used formal verification technique, phase abstraction is considered one of the most commonly used preprocessing techniques. We present an approach to certify an extended form of phase abstraction using a generic certificate format. As in earlier works our approach involves constructing a witness circuit with an inductive invariant property that certifies the correctness of the entire model checking process, which is then validated by an independent certificate checker. We have implemented and evaluated the proposed approach including certification for various preprocessing configurations on hardware model checking competition benchmarks. As an improvement on previous work in this area, the proposed method is able to efficiently complete certification with an overhead of a fraction of model checking time.

9.1 Introduction

Over the past few decades, symbolic model checking [33, 234, 235] has been put forward as one of the most effective techniques in formal verification. A lot of trust is placed into model checking tools when assessing the correctness of safety-critical systems. However, model checkers themselves and the symbolic reasoning tools they rely on, are exceedingly complex, both in the theory of their algorithms and their practical implementation. They often run for multiple days, distributed across hundreds of interacting threads, ultimately yielding a single bit of information signaling the verification result. To increase trust in these tools, several approaches have attempted to implement fully verified model checkers in a theorem proving environment such as Isabelle [236–238]. However, the scalability as well as versatility of those tools is often rather limited. For example, a technique update tends to require the entire tool to be re-verified.

An alternative is to make model checkers provide machine-checkable proofs as certificates that can be validated by independent checkers [239–246], which is already a successful approach in SAT [133, 134], i.e., proofs are mandatory in the SAT competition since 2016 [136], and they are a very hot topic in SMT [247–250] and beyond [247]. Crucially, these certificates need to be simple enough to allow the implementation of a fully verified proof checker [138, 251, 252], and preferably verifiable "end-to-end", i.e., certifying all stages of the model checking process, including all forms of preprocessing steps.

The approach in [12, 13, 253] introduces a generic certificate format that can be directly generated from hardware model checkers via book-keeping. More specifically, the certificate is in the form of a Boolean circuit that comes with an inductive invariant, such that it can be verified by six simple SAT checks. So far, it has shown to be effective across several model checking techniques, but has not covered phase abstraction [223]. The experimental results from [12, 13, 253] also show performance challenges with more complex model checking problems. In this paper, we focus on refining the format for smaller certificates while accommodating additional techniques such as cone-of-influence analysis reduction [235].

Phase abstraction [223] is a popular preprocessing technique which tries to simplify a given model checking problem by detecting and removing periodic signals that exhibit clock-like behaviors. These signals are essentially the clocks embedded in circuit designs, often due to the design style of multi-phase clocking [254]. Phase abstraction helps reduce circuit complexity therefore making the backend model checking task easier. Differently from [255, 256] where the concept was first suggested, requiring syntactic analysis and user inputs, phase abstraction [223] makes use of ternary simulation to automatically identify a group of clock-like latches. Beside this, ternary simulation has also been utilized in the context of temporal decomposition [257] for detecting transient signals.

In industrial settings, due to the use of complex reset logic as well as circuit synthesis optimizations, clock signals are sometimes delayed by a number of initialization steps [225]. To further optimize the verification procedure we extend phase abstraction by exploiting the power of ternary simulation to capture different classes of periodic signals including those that are considered partially as clocks as well as equivalent signals [258]. An optimal phase number is computed based on globally extracted patterns, which then is used to unfold the circuit multiple times. The resulting unfolded circuit further undergoes rewriting and cone-of-influence reduction, before it is passed on to a base model checker for final verification. To summarize our contributions are as follows:

1. We formalize, revisit and extend the original phase abstraction [223] by introducing periodic signals, that are then identified and removed for circuit reduction. Our technique also subsumes temporal decomposition [257].
2. Building upon [12, 13, 253], we propose a refined certificate format for hardware model checking based on a new *restricted simulation* relation. We demonstrate how to build such a certificate for extended phase abstraction.
3. We present MC2, a certifying model checker that implements our proposed preprocessing technique and generates certificates for the entire model checking process. We show empirically that the approach requires small certification overhead in contrast to [12, 13, 253].

After background in Section 9.2, Section 9.3 introduces the notion of periodic signals. In Section 9.4 we present an extended variant of phase abstraction that simplifies the original model with periodic signals. In Section 9.5 we define a refined certificate format and present a general certification approach for phase abstraction. In Section 9.6 we describe the implementation of MC2 and then show the effectiveness of our new certification approach in Section 9.7.

9.2 Background

Given a set of Boolean variables $\mathcal{V}$, a literal l is either a variable $v \in \mathcal{V}$ or its negation $\neg v$. A *cube* is considered to be a non-contradictory set of literals. Let c be such a cube over a set of variables L and assume L' are copies of L, i.e., each $l \in L$ corresponds bijectively to an $l' \in L'$. Then we write $c(L')$ to denote the resulting cube after replacing the variables in c with its corresponding variables in L'. For a Boolean formula f, we write $f|_l$ and $f|_{\neg l}$ to denote the formula after substituting all occurrences of the literal l with $\top$ and $\bot$ respectively. We use equality symbols $\simeq$ [259] and $\equiv$ to denote syntactic and semantic equivalence and similarly $\rightarrow$ and $\Rightarrow$ to denote syntactic and semantic logical implication.

Definition 9.1 Circuit. A circuit C is represented by a quintuple (I, L, R, F, P), where I and L are (finite) sets of input and latch variables. The reset functions are given as $R = \{r_l(I, L) \mid l \in L\}$ where the individual reset function $r_l(I, L)$ for a latch $l \in L$ is a Boolean formula over inputs I and latches L. Similarly the set of transition functions is given as $F = \{f_l(I, L) \mid l \in L\}$. Finally $P(I, L)$ denotes a safety property corresponding to set of good states again encoded as a Boolean formula over the inputs and latches.

This notion can be extended to more general circuits involving for instance word-level semantics or even continuous variables by replacing in this definition Boolean formulas by corresponding predicates and terms in first-order logic modulo theories. For simplicity of exposition we focus in this work on Boolean semantics, which matches the main application area we are targeting, i.e., industrial-scale gate-level hardware model checking. We claim that extensions to "circuits modulo theories" are quite straightforward.

A concrete state is an assignment to variables $I \cup L$. Therefore the set of reset states of a circuit is the set of satisfying assignments to $R(L) = \bigwedge_{l \in L} (l \simeq r_l(I, L))$. Note the use of syntactic equality "$\simeq$" in this definition.

As in previous work [12] we assume acyclic reset functions. Therefore $R(L)$ is always satisfiable. A circuit with acyclic reset functions is called *stratified*.

As in bounded model checking [260], with I_i and L_i "temporal" copies of I and L at time step i, the *unrolling* of a circuit up to length k is expressed as:

$$U_k = \bigwedge_{i \in [0,k)} (L_{i+1} \simeq F(I_i, L_i)).$$

Cube simulation [13] subsumes ternary simulation such that a lasso found by ternary simulation can also be found via cube simulation. A cube simulation is a sequence of cubes $c_0, \ldots, c_\delta, \ldots, c_{\delta+\omega}$ over latches L such that (1) $R(L) \Rightarrow c_0$; (2) $c_i \wedge (L' \simeq F(I, L)) \Rightarrow c'_{i+1}$ for all $i \in [0, \delta + \omega)$, where c'_{i+1} is the primed copy of c_{i+1}. It is called a cube lasso if $c_{\delta+\omega} \wedge (L' \simeq F(I, L)) \Rightarrow c'_\delta$. In which case δ is the stem length and ω is the loop length. For $\delta = 0$, the initial cube is already part of the loop and for $\omega = 0$, the lasso ends in a self-loop.

9.3 Periodic Signals

In sequential hardware designs, signals that eventually stabilize to a constant, i.e., to $\top$ or $\bot$, after certain initialization steps are called *transient* signals [13, 257], whereas oscillating signals have clock-like or periodic behaviors. A simplest example of a clock is a latch that always oscillates between $\top$ and $\bot$.

Since hardware designs typically consist of complex initialization logic, there are occurrences of delayed oscillating signals, like clocks that start ticking after several reset steps, with a combination of transient and clock behaviours. We generalize this concept to categorize latches as periodic signals associated with a *duration* (i.e., the number of time steps for which a signal is delayed) and a *phase number* (i.e., the period length in a periodic behavior). Moreover, our generalization also captures equivalent and antivalent signals [258], as well as those that exhibit partial periodic behaviours. See Fig. 9.1 for an example.

Definition 9.2 Periodic Signal**.** Given a circuit $C = (I, L, R, F, P)$ and a cube lasso $c_0, \ldots c_\delta, \ldots, c_{\delta+\omega}$. A periodic signal λ_l for a latch $l \in L$ is defined as $\lambda_l = (d, [v^0, \ldots, v^{n-1}])$ where $d \in \mathbb{N}$, $n \in \mathbb{N}^+$ and v^i is a latch literal or a constant, with $d \leq \delta$. We further

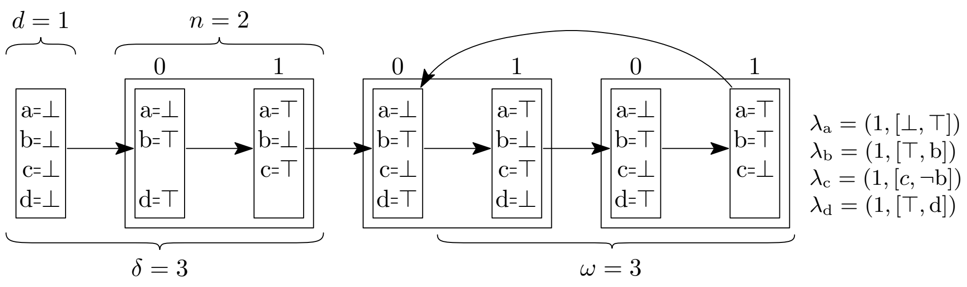

Figure 9.1: An example of a cube lasso over the latches $l \in L = \{a, b, c, d\}$. In the example the tall rectangles represent cubes as partial assignments, i.e., the second cube from the left is $(\neg a) \wedge b \wedge d$. Phase 0 and 1 are marked on top of the cubes. As shown, duration $d = 1$ and phase number $n = 2$ yield a high number of useful signals for this cube lasso. Each latch l is associated with a periodic pattern λ_l on the right describing its behaviors for phase 0 and 1. If a latch is missing from a cube for a certain phase, it has no constraint thus we use the equality of the latch to itself in the signal. Latch a turns out to be a simple clock delayed by one step. Latches b and d behave clock-like but only in phase 0. Latch c always has the opposite value of latch b in phase 1. Note that we could also have $\neg c$ in phase 1 of signal λ_b but choosing a single representative for a set of equivalent signals is beneficial for the following simplification steps.

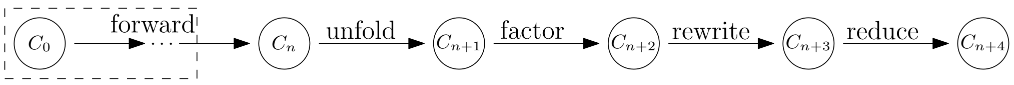

Figure 9.2: Circuit transformation using phase abstraction.

require that there exist $k^0, k^1 \in \mathbb{N}^+$ with $k^0 \cdot n + d = \delta$ and $k^1 \cdot n = \omega + 1$ such that for all $i \in [0, n)$ and $j \in [0, k^0 + k^1)$ we have $c_{i+j\cdot n} \Rightarrow (l \simeq v^i)$.

For a signal $\lambda_l = (d, [v^0, \ldots, v^{n-1}])$ we will write λ_l^i to refer to the i-th element of $[v^0, \ldots, v^{n-1}]$, which we refer to as its phase. See Fig. 9.1 for an example where $k^0 = 1$ and $k^1 = 2$.

9.4 Extending Phase Abstraction

In this section, we revisit and extend phase abstraction by defining it as a sequence of preprocessing steps, as illustrated in Fig. 9.2. Differently from the approach in [223], we present phase abstraction as part of a compositional framework, that handles a more general class of periodic signals. As our approach subsumes temporal decomposition adopted from the framework in [13], we first apply *circuit forwarding* [13] for duration d (i.e., unrolling the reset states of a circuit by d steps) before unfolding is performed.

Fig. 9.2 illustrates the flow of phase abstraction. The process begins by using cube simulation to identify a set of periodic signals as defined in Section 9.3 and computing an optimal duration and phase number based on a selected cube lasso. Once the circuit is unfolded n times, factoring is performed by assigning constant values to the clock-like

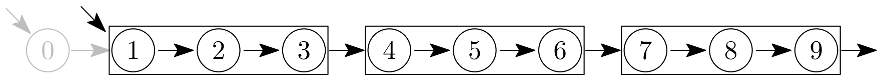

Figure 9.3: An example of a forwarded ($d = 1$) and unfolded ($n = 3$) circuit. The circles denote states in the original circuit (0 is the initial state). The rectangle are states in the unfolded circuit.

signals as well as replacing latches with their equivalent or antivalent representative latches in each phase. Next, the property is rewritten by applying structural rewriting techniques and the circuit is further simplified using cone-of-influence reduction. Finally, the simplified circuit (C_{n+4} in Fig. 9.2) is checked using a base model checking approach such as IC3/PDR [49] or continue to be preprocessed further.

In Fig. 9.3, we show intuitively an example of a circuit with 4-bit states representing 0,...,9 and so on, where the initial state is 0. After forwarding the circuit by one step ($d = 1$), the initial state becomes 1. Subsequently with an unfolding of $n = 3$, every state (marked with rectangles) in the unfolded circuit consists of three states from the original circuit. We introduce the formal definitions below.

Unfolding a circuit simply means to copy the transition function multiple times to compute n steps of the original circuit at once. Each copy of the transition function only has to deal with a single phase and can therefore be optimized.

Definition 9.3 Unfolded circuit**.** Given a circuit $C = (I, L, R, F, P)$ and a phase number $n \in \mathbb{N}^+$. The unfolded circuit $C' = (I', L', R', F', P')$ is:

1. $I' = I^0 \cup \cdots \cup I^{n-1}$; $L' = L^0 \cup \cdots \cup L^{n-1}$.
2. $R' = \{r'_l \mid l \in L'\}$: for $l \in L^0, r'_l = r_l$;
   for $i \in (0, n), l^i \in L^i, r'_{l^i} = F(I^i, L^{i-1})$.
3. $F' = \{f'_l \mid l \in L'\}$: for $l \in L^0, f'_l = f_l(I^0, L^{n-1})$;
   for $i \in (0, n), l^i \in L^i, f'_{l^i} = f_{l^i}(I^i, L^{i-1})$.
4. $P' = \bigwedge_{i \in [0,n)} P(I^i, L^i)$.

We obtain a simplified circuit by replacing the periodic signals with constants and equivalent/antivalent latches in the unfolded circuit.

Definition 9.4 Factor circuit**.** For a fixed duration d and phase number n, given a d-forwarded and n-unfolded circuit $C = (I, L, R, F, P)$ and a periodic signal with duration d and phase number n for each latch, the factor circuit $C' = (I, L, R', F', P)$ is defined by:

$R' = \{r'_l \mid l \in L\}$:

- $r'_{l^i} = \lambda^i_l$, if $\lambda^i_l \in \{\bot, \top\}$;
- $r'_{l^i} = r_{\lambda^i_l}$, if $\lambda^i_l \in L$.
- $r'_{l^i} = \neg r_{\neg\lambda^i_l}$, otherwise.

$F' = \{f'_l \mid l \in L\}$:

- $f'_{l^i} = \lambda^i_l$, if $\lambda^i_l \in \{\bot, \top\}$;
- $f'_{l^i} = f_{\lambda^i_l}$, if $\lambda^i_l \in L$.
- $f'_{l^i} = \neg f_{\neg\lambda^i_l}$, otherwise.

Replaced latches will be removed by a combination of rewriting and cone-of-influence reduction introduced in the following definitions. There are various rewriting techniques also including SAT sweeping [57, 261–265].

Definition 9.5 Rewrite circuit**.** Given a circuit $C = (I, L, R, F, P)$, a rewrite circuit $C' = (I, L, R, F, P')$ satisfies $P \equiv P'$.

For a given circuit, we apply cone-of-influence reduction to obtain a reduced circuit such that latches and inputs outside the cone of influence are removed.

Definition 9.6 Reduced circuit**.** Given a circuit $C = (I, L, R, F, P)$. The reduced circuit $C' = (I', L', R', F', P)$ is defined as follows:

- $I' = I \cap coi(P)$;
- $L' = L \cap coi(P)$;
- $R' = \{r_l \mid l \in L'\}$;
- $F' = \{f_l \mid l \in L'\}$,

where the cone of influence of the property $coi(P) \subseteq (I \cup L)$ is defined as the smallest set of inputs and latches such that $vars(P) \subseteq coi(P)$ as well as $vars(r_l) \subseteq coi(P)$ and $vars(f_l) \subseteq coi(P)$ for all latches $l \in coi(P)$.

9.5 Certification

We define a revised certificate format that allows smaller and more optimized certificates. We then propose a method for producing certificates for phase abstraction. The proofs for our main theorems can be found in the Appendix.

9.5.1 Restricted simulation

In the following, we define a new variant of the stratified simulation relation [12], which we call *restricted simulation*, that considers the intersection of latches shared between two given circuits as a common component.

Definition 9.7 Restricted Simulation**.** Given stratified circuits C' and C with $C' = (I', L', R', F', P')$ and $C = (I, L, R, F, P)$. We say C' *simulates* C under the *restricted simulation relation* iff

1. For $l \in (L \cap L'), r_l(I, L) \equiv r'_l(I', L')$.
2. For $l \in (L \cap L'), f_l(I, L) \equiv f'_l(I', L')$.
3. $P'(I', L') \Rightarrow P(I, L)$.

This new simulation relation differs from [12, 253], where inputs were required to be identical in both circuits ($I = I'$), and latches in C had to form a subset of latches in C' ($L \subseteq L'$). Therefore, under those previous "combinational" [253] or "stratified" [12] simulation relations the simulating circuit C' cannot have fewer latches than L. This is a feature we need for instance when incorporating certificates for cone-of-influence

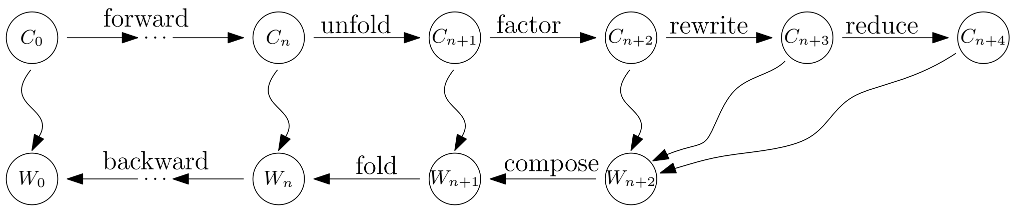

Figure 9.4: Certification for (extended) phase abstraction. Base model checking is performed on circuit C_{n+4}, which produces a witness circuit W_{n+2}, that certifies $C_{n+2}, C_{n+3},$ and C_{n+4}. We construct step-wise to obtain W_0, which is a certificate for the entire model checking procedure.

reduction [235], a common preprocessing technique. It opens up the possibility to reduce certificate sizes substantially.

Still, as for stratified simulation, restricted simulation can be verified by three simple SAT checks, i.e., separately for each of the three requirements in Def. 9.7.

Definition 9.8 Semantic independence**.** Let $\mathcal{V}$ be a set of variables and $v \in \mathcal{V}$. Then a formula $f(\mathcal{V})$ is said to be semantically independent of v iff

$$f(\mathcal{V})|_v \equiv f(\mathcal{V})|_{\neg v}.$$

Semantic dependency [266–269] allows us to assume that a formula only depends on a subset of variables, which without loss of generality simplifies proofs used for the rest of this section. The stratified assumption for reset functions entails no cyclic dependencies thus $R'(L')$ is satisfiable. A reset state in a circuit is simply a satisfying assignment to the reset predicate $R(L)$. Based on the reset condition (Def. 9.7.1), it is however necessary to show that for every reset state in C it can always be extended to a reset state in C', while the common variables have the same assignment in both circuits. This is stated in the lemma below, and the proofs can be found in the Appendix.

Lemma 9.9. *Let $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ be two stratified circuits satisfying the reset condition defined in Def. 9.7.1. Then $R'(L \cap L')$ is semantically dependent only on their common variables.*

In fact, semantic independence is a direct consequence of restricted simulation; thus no separate check is required. We make a further remark that if the reset function is dependent on an input variable, then it has to be an input variable common to both circuits.

Based on this, we conclude with the main theorem for restricted simulation such that C is safe if C' is safe (i.e., no bad state that violates the property is reachable from any initial state).

Theorem 9.10. *Let $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ be two stratified circuits, where C' simulates C under restricted simulation.*
If C' is safe, then C is also safe.

Intuitively, if there is an unsafe trace in C, Def. 9.7.1 together with Lemma 9.9 allow us to find a simulating reset state and transition it with Def. 9.7.2 to a simulating state also violating the property in C' by Def. 9.7.3. Here a state in C' simulates a state in C if they match on all common variables. Building on this, we present witness circuits as a format for certificates. Verifying the restricted simulation relation requires three SAT checks, and another three SAT checks are needed for validating the inductive invariant [253]. Therefore certification requires in total six SAT checks as well as a polynomial time check for reset stratification.

Definition 9.11 Witness circuit**.** Let $C = (I, L, R, F, P)$ be a stratified circuit. A witness circuit $W = (J, M, S, G, Q)$ of C satisfies the following:

- W simulates C under the restricted simulation relation.
- Q is an inductive invariant in W.

The witness circuit format subsumes [12, 13], thus every witness circuit in their format is also valid under Def. 9.11.

9.5.2 Certifying Phase Abstraction

The certificate format is generic, subsumes [13], and is designed to potentially be used as a standard in future hardware model checking competitions. We proceed to demonstrate how a certificate can be constructed for a model checking pipeline that includes phase abstraction. The theorems in this section state that this construction guarantees that a certificate will be produced. We illustrate our certification pipeline in Fig. 9.4. After phase abstraction and base model checking, we can build a certificate backwards based on the certificate produced by the base model checker. The following theorem states that the witness circuit of the reduced circuit serves as a witness circuit for the original circuit too.

Theorem 9.12. *Given a circuit $C = (I, L, R, F, P)$ and its reduced circuit $C' = (I', L', R', F', P')$. A witness circuit of C' is also a witness circuit of C.*

The outcome of rewriting is a circuit with a simplified property that maintains semantic equivalence with the original property. Therefore in our framework, the certificate for the simplified property is also valid for the original property. Furthermore, certificates can be optimized by rewriting at any stage. We summarize this in the following proposition.

Proposition 9.13. *Given a circuit C and its rewrite circuit C'. A witness circuit of C' is also a witness circuit of C.*

We define the composite witness circuit to combine the certificates for cube simulation and the factor circuit.

Definition 9.14 Composite witness circuit**.** Given a stratified circuit $C = (I, L, R, F, P)$ and its factor circuit $C' = (I', L', R', F', P')$, and the unfolded loop invariant $\phi =$

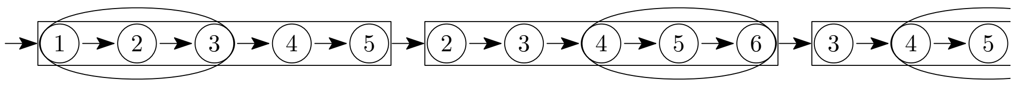
Figure 9.5: Every fully initialized state of a 3-folded witness circuit contains 3 original states that form an unfolded state. Two consecutive 3-folded states contain either the same unfolded states or two states consecutive in the unfolded circuit.

$\vee_{i\in[0,m)} \wedge_{j\in[0,n)} c_{i*n+j+d}$, with $m = (\delta + \omega - d + 1)/n$, obtained from the cube lasso. Let $W' = (J', M', S', G', Q')$ be a witness circuit of C'. The composite witness circuit $W = (J, M, S, G, Q)$ is defined as follows:

1. $J = I \cup J'$.
2. $M = L \cup (M' \backslash L')$.
3. $S = \{s_l \mid l \in M\}$:
   a) for $l \in L, s_l = r_l$;
   b) for $l \in M' \backslash L', s_l = s'_l$.
4. $G = \{g_l \mid l \in M\}$:
   a) for $l \in L, g_l = f_l$;
   b) for $l \in M' \backslash L', g_l = g'_l$.
5. $Q = \phi(L) \wedge Q'(J', M')$.

Theorem 9.15. *Given circuit $C = (I, L, R, F, P)$, and factor circuit $C' = (I', L', R', F', P')$. Let $W' = (J', M', S', G', Q')$ be a witness circuit of C', and $W = (J, M, S, G, Q)$ constructed as in Def. 9.14. Then W is a witness circuit of C.*

In the construction of an n-folded witness circuit from the unfolded witness W', a single instance of W''s latches (N), yet multiples of the original latches L are used. As illustrated in Fig 9.5, these L record a history, contrasting with their role in the unfolded circuit where they calculate multi-step transitions.

Definition 9.16 n-folded witness circuit**.** Given a circuit $C = (I, L, R, F, P)$ with a phase number $n \in \mathbb{N}^+$, and its unfolded circuit $C' = (I', L', R', F', P')$. Let $W' = (J', M', S', G', Q')$ be the witness circuit of C'. The n-folded witness circuit $W = (J, M, S, G, Q)$ is defined as follows:

1. $J = I^0 \cup J^0$, where I^0 and J^0 are I and J' respectively.
2. $M = I^1 \cdots I^m \cup L^0 \cdots L^m \cup N \cup J^1 \cup \{b^0 \cdots b^m, e^0 \cdots e^{n-2}\}$, where $m = 2 \times n - 2$, $N = M' \setminus L'$, and I^i, L^i are copies of I and L, and J^1 is a copy of J'.
3. $S = \{s_l \mid l \in M\}$:
   a) $s_{b^0} = \top$;
   b) For $i \in (0, m], s_{b^i} = \bot$.
   c) For $i \in [0, n-1), s_{e^i} = \bot$.
   d) For $l \in L^0, s_l = r'_l$.
   e) For $l \in (I^1 \cdots I^m \cup L^1 \cdots L^m \cup J^1), s_l = l$.

f) For $l \in N, s_l = s'_l$.

4. $G = \{g_l \mid l \in M\}$:

a) $g_{b^0} = \top$.

b) For $i \in [1, m], g_{b^i} = b^{i-1}$.

c) $g_{e^0} = b^{n-1} \wedge \neg e^{n-2}$.

d) For $i \in [1, n-1), g_{e^i} = e^{i-1} \wedge \neg e^{n-2}$.

e) For $l \in L^0, g_l = f_l$.

f) For $l^1 \in J^1, g_{l^1} = l^0$.

g) For $i \in [1, m], l^i \in (I^i \cup L^i), g_{l^i} = l^{i-1}$.

h) For $l \in N$,
$g_l = ite(e^{n-2}, g'_l(J^1, M' \cap (I^{m-n+1} \cdots I^m \cdots L^{m-n+1} \cdots L^m \cup N)), l)$.

5. $Q = \bigwedge_{i \in [0,6]} q^i$:

a) $q^0 = P(I^0, L^0)$.

b) $q^1 = b^0$.

c) $q^2 = \bigwedge_{i \in [1,m]} (b^i \to b^{i-1})$.

d) $q^3 = \bigwedge_{i \in [1,m]} (b^i \to (L^i \simeq F(I^{i-1}, L^{i-1})))$.

e) $q^4 = \bigwedge_{i \in [1,m]} ((\neg b^i \wedge b^{i-1}) \to (R(L^{i-1}) \wedge S'(N)))$.

f) $q^5 = b^m \to (\bigvee_{i \in [0,n)} ((\bigwedge_{j \in [i,n-1)} \neg e^j) \wedge (\bigwedge_{j \in [0,i)} e^j) \wedge$
$Q'(J^0, M' \cap (L^i \cdots \cup L^{i+n-1} \cup N)))$.

g) $q^6 = \bigwedge_{i \in [1,n-2]} (e^i \to e^{i-1})$.

h) $q^7 = \bigwedge_{i \in [0,n-2]} (e^i \to b^{n+i})$.

i) $q^8 = \bigwedge_{i \in [0,n-2)} ((\neg b^m \wedge b^{n+i}) \to e^i)$.

The b^is are used for encoding initialization. So that inductiveness is ensured when not all copies are initialized. The $n-1$ bits e^i are used to determine which set of n consecutive original states form an unfolded state (a state in the unfolded circuit). This information is used to determine on which copies the unfolded property needs to hold and to transition the latches in N (the part of the witness circuit added by the backend model checker) once every n steps.

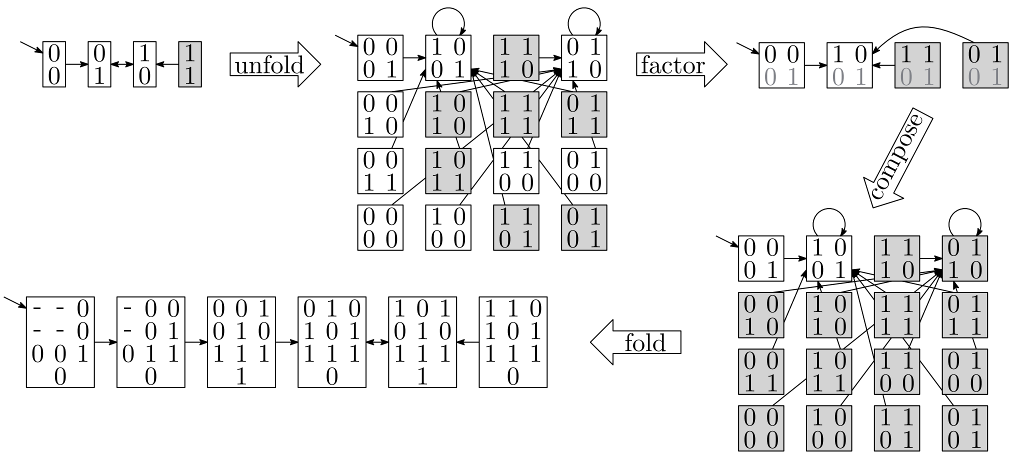

Figure 9.6: A concrete example of the model checking and certification pipeline. The original circuit has two latches; the bottom latch alternates and the top copies the previous value of this clock. The property is that at least one bit is unset. Bad states are marked gray. After unfolding with phase number two, the size of the state space is squared. Since the bottom bit is periodic, we can replace it with a constant in each phase (factor). On this circuit terminal model checking is performed, since the property is already inductive (no transition from good to bad), the circuit serves as its own witness. To produce the final witness circuit, the clock is added back as a latch, and the property is extended with the loop invariant asserting that the clock has the correct value for each phase. Lastly, the circuit is *folded* to match the speed of the original circuit. Three initialization bits b^i are introduced and one additional bit e^0 that determines which pair of consecutive states need to fulfill the property (0 for the right pair and 1 for the left). This check is only part of the property once full initialization is reached. For this final witness circuit, only the good states are depicted. Also, the first two states represent sets of good states with the same behavior.

Theorem 9.17. *Given a circuit $C = (I, L, R, F, P)$ with a phase number $n \in \mathbb{N}^+$, its unfolded cicuit $C' = (I', L', R', F', P')$ with a witness circuit $W' = (J', M', S', G', Q')$. Let $W = (J, M, S, G, Q)$ be the circuit constructed as in Def. 9.16. Then W is a witness circuit of C.*

After the witness circuit has been folded, the same construction from [13] can be used to construct the backward witness. With that, the pipeline outlined in Fig. 9.4 is completed. If phase abstraction is the first technique applied by the model checker, a final witness is obtained. Otherwise, further witness processing steps still need to be performed.

9.6 Implementation

In this section, we present MC2, a certifying model checker implementing phase abstraction and IC3. We implement our own IC3 since no existing model checker supports reset functions or produces certificates in the desired format. We used fuzzing to increase trust in our tool. The version of MC2 used for the evaluation, was tested on over 25 million randomly generated circuits [51] in combination with random parameter configurations. All produced certificates where verified.

To extract periodic signals we perform ternary simulation [228] while using a forward-subsumption algorithm based on a one-watch-literal data structure [123] to identify supersets of previously visited cubes, and thereby a set of cube lassos. For each cube lasso we consider every factor of the loop length ω as a phase number candidate n. We also consider every duration d, that renders the leftover tail length ($\delta - d$) divisible by n. To keep the circuit sizes manageable, we limit both n and d to a maximum of 8. We call each pair (d, n) an unfolding candidate and compute the corresponding periodic signal (Def. 9.2) for each latch.

For each phase, equivalences are identified by inserting a bit string corresponding to the signs of each latch into a hash table. After identifying the signals, forwarding and unfolding are performed on a copy of the circuit, followed by rudimentary rewriting. Currently the rewriting does not include structural hashing and is mostly limited to constant propagation. Afterwards a sequential cone-of-influence analysis starting from the property is performed. After performing these steps for each candidate, we pick the duration-phase pair that yields a circuit with the fewest latches and give it to a backend model checker.

We evaluated the preprocessor on three backend model checkers: the open-source k-induction-based model checker McAiger [270](Kind in the following), the state-of-the-art IC3 implementation in ABC [177] and our own version of IC3 that supports reset functions and produces certificates. Since ABC does not support reset functions, it is not able to model check any forwarded circuit (note that implementing this feature on ABC is also a non-trivial task), therefore for this configuration we only ran phase abstraction without forwarding thus no temporal decomposition.

Our IC3 implementation on MC2 does feature cube minimization via ternary simulation [224], however it is missing proof-obligation rescheduling. In fact, we currently use a simple stack of proof obligations as opposed to a priority queue. Despite using one SAT solver instance per frame, we also do not feature cones-on-demand, but instead always translate the entire circuit using Tseitin [36].

Lastly, we also modified the open source implementation of Certifaiger [271] to support certificates based on restricted simulation. For a witness circuit C' of C, the new certificate checker encodes the following six checks as combinatorial AIGER circuits and then uses the `aigtocnf` to translate them to SAT:

Ⓐ The property of C' holds in all initial states.

Ⓑ The property of C' implies the property for successor states.

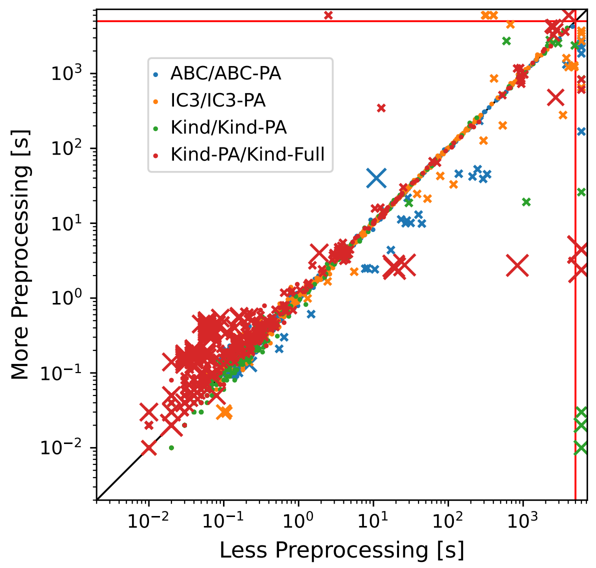

Figure 9.7: Comparison of model checking performance. We compare four pairs of configurations; the three backend engines with and without phase abstraction (with fixed duration 0) and for Kind we present the effect of additionally allowing forwarding. The size of the markers represents $n + d$. The dots represent instances where the preprocessing heuristic decided not to alter the circuit. The red lines mark the timeout of 5000 seconds. Markers beyond that line represent instances solved by one configuration but not the other.

Ⓒ The property of C' holds in all good states.

Ⓓ The reset functions of common latches are equivalent. (Def. 9.7. 1)

Ⓔ The transition functions of common latches are equivalent. (Def. 9.7. 2)

Ⓕ The property of C' implies the property of C. (Def. 9.7. 3)

The first three checks are unchanged and encode the standard check for P' being an inductive invariant in C'. Since P' is both the inductive invariant and the property we are checking, Ⓒ can technically be omitted. However, in our implementation, the inductiveness checker is an independent component from the simulation checker and would also works for scenarios where the inductive invariant is a strengthening of the property in C'.

9.7 Experimental Evaluation

This section presents experimental results for evaluating the impact of preprocessing on the different backends, as well as the effectiveness of our proposed certification approach. The experiments were run in parallel on a cluster of 32 nodes. Each node was

	Model				ABC		Our IC3			Kind		
	Safe	$\overline{n}$	d	n		PA		PA	Full		PA	Full
Solved					740	**745**	715	715	604	533	538	544
PAR2					996	**941**	1357	1351	2699	3533	3472	3399
abp4p2ff	✓	1	1	1	1.12	**1.08**	6.35	6.23	6.18	2.50	2.50	
bjrb07		3	0	3	0.12	0.10	0.11	**0.03**	0.07		**0.03**	0.04
nusmvb5p2		5	0	5	0.12	0.10	**0.01**	**0.01**	**0.01**		**0.01**	**0.01**
nusmvb10p2		5	0	5	0.22	0.13	0.10	0.03	0.04		**0.02**	**0.02**
prodcell0	✓	1	5	8	26.97	27.07	228.46	243.73	49.76			**2.37**
prodcell0neg	✓	1	5	8	16.36	15.93	230.57	230.67	36.62			**2.39**
prodcell1	✓	1	7	8	23.45	23.38	654.21	665.86	59.67			**4.43**
prodcell1neg	✓	1	7	8	28.36	28.33	681.11	738.61	61.74			**4.48**
prodcell2	✓	1	7	8	24.98	24.58	661.71	663.37	56.74			**4.43**
prodcell2neg	✓	1	7	8	20.23	20.28	778.39	768.75	56.14			**4.47**
bc57sen0neg	✓	1	1	1	503.61	**494.55**	910.72	906.87	1760.41			830.92
abp4ptimo	✓	1	1	1	4.14	**4.13**	28.93	29.91	6.32			608.55
boblivea		1	2	1	3.70	**3.68**			7.85			
bobsm5378d2		1	8	1	**4.04**	4.12			88.36			
bobsmnut1		8	5	8	**10.95**	40.08			2504.07			
prodcell3neg	✓	2	2	8	27.88	10.86	310.22				837.43	**2.73**
prodcell4neg	✓	2	2	8	44.31	9.90	404.12				26.04	**2.77**
prodcell3	✓	2	2	8	23.45	11.24	320.23			1103.29	19.22	**2.48**
prodcell4	✓	2	2	8	31.40	10.08	398.83			29.71	18.67	**2.68**
pdtvisvsar29		1	2	5					1523.73	0.36	**0.29**	0.40
intel042	✓	1	3	2						**3876.04**	4061.38	
intel022		2	2	2		**1852.29**						
intel021		2	2	2		2752.86		**651.56**				
intel023		2	2	2		**2257.94**		3728.38				
intel029		2	2	2		**2550.14**		3437.64				
intel024		2	2	2		**167.96**	676.64	4526.60				
intel019		2	2	2				**2716.40**				

Table 9.1: We presents the effect of preprocessing in combination with different backend engines on model checking time. We compare no preprocessing to only phase abstraction without forwarding (PA) and full preprocessing (Full). Note that, ABC does not support reset functions and can therefore not be combined with full preprocessing. For each model we present the phase number without forwarding $\overline{n}$ for PA and the duration d and phase number n corresponding to Full. Models where the property holds are marked as safe. The first two rows present the number of solved instances and the PAR2 score [14] over all 818 benchmarks. The table shows all instances where preprocessing had either a positive or negative impact on model checking success, with the exception of those instances rendered unsolvable for our IC3 implementation by forwarding.

equipped with two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz and 128 GB of main memory. We allocated 8 instances to each node, with a timeout of 5000 seconds for model checking and 10 000 seconds for certificate checking. Memory is limit to 8 GB per instance in both cases.

The benchmarks are obtained from HWMCC2010 [272] which contains a good number of industrial problems. As we observe from the experiments in general, preposessing is usually fast. Ignoring one outlier in our benchmark set, it completes within an average of 0.07 seconds and evaluates no more than 17 unfolding candidates per

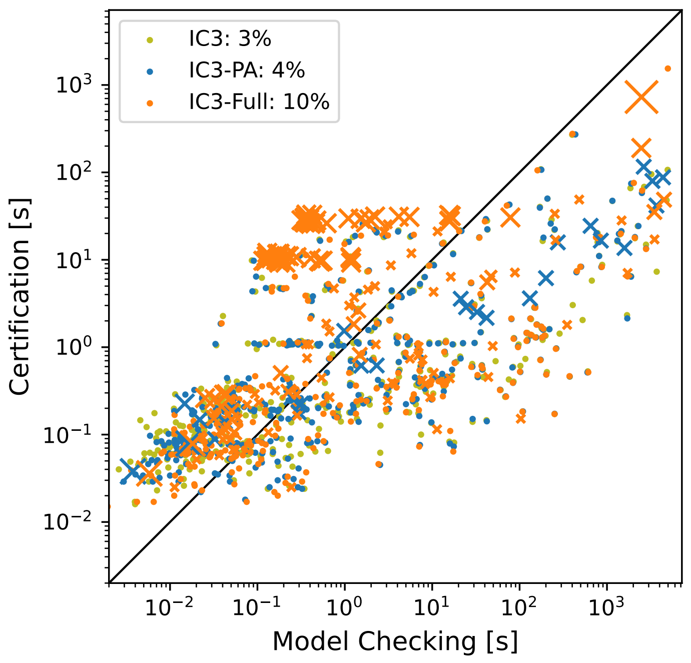

Figure 9.8: Certification vs. model checking time for three configurations of our IC3 engine. The legend shows the cumulative overhead of including certification for all solved instances. The size of the markers represents $n + d$. The dots represent instances where preprocessing did not alter the circuit.

benchmark. Interestingly, for the outlier "bobsmnut1", 3019 unfoldings are computed for 179 different cube lassos within 34 seconds.

Table 9.1 presents the effect of our preprocessing on different backends, further illustrated in Fig 9.7. Our preprocessor was able to improve the performance of the sophisticated IC3/PDR implementation in ABC, allowing us to solve five more instances, all from the intel family. For each benchmark from this family, our heuristic computed an optimal phase number of 2. A likely explanation for this is that the real-world industrial designs tend to contain strict two-phase clocks [255]. The positive effect of phase abstraction is also clear in combination with the k-induction (Kind) backend. Circuit forwarding provides a further improvement, that is especially notable on the prodcell benchmarks. These also illustrate how forwarding enables more successful unfolding. Without forwarding, preprocessing only unfolds 61 out of the 818 benchmarks with an average phase number of 2, with forwarding 152 circuits are unfolded with an average phase number of 4.

Even though our prototype implementation of IC3 is missing a number of important features present in ABC, it still solves a large number of benchmarks. However, as opposed to ABC it does lose a number of benchmarks with phase abstraction. This can be explained by the lack of sophisticated rewriting that can exploit the unfolded circuits structure. The addition of forwarding is highly detrimental to performance, losing 115 instances. This is due to our implementation following the PDR design outlined in [224]. It requires any blocked cube not to intersect the initial states after generalization. If only a single reset state exists this check is linear in the size of the cube. However, in the presence of reset functions it is implemented with a SAT call. While also slower the main problem however is that the reset-intersection check is also more likely to block generalization. On the 115 lost benchmarks generalization failed 96% of the time, while it only failed in 1.8% of the cases without forwarding. We keep the optimization of our IC3 implementation in the presence of reset functions for future work.

Fig. 9.8 displays certification results on MC2 in comparison to model checking time. IC3 provides certificates that are easily verifiable, as confirmed by our experiments with cumulative overhead of only 3%. The addition of phase abstraction (i.e., including constructing n-folded witnesses as in Fig. 9.4, without witness back-warding) does not bring significant additional overhead. When forwarding is allowed, the certification overhead increases to 10%. The run time of certificates generation and encoding to SAT is negligible for all configurations. The certification time is dominated by the SAT solving time for the transition (Def. 9.7.2) and consecution check. Overall, this is a significant improvement over related work from [13] which reported 1154% overhead on the same set of benchmarks using a k-induction engine as the backend.

9.8 Conclusion

In this paper, we present a certificate format that can be effectively validated by an independent certificate checker. We demonstrate its versatility by applying it to an extended version of phase abstraction, which we introduce as one of the contributions of this paper. We have implemented the proposed approach on a new certifying model checker MC2. The experimental results on HWMCC instances show that our approach is effective and yields very small certification overhead, as a vast improvement over related work. Our certificate format allows for smaller certificates and is designed to be possibly used in hardware model checking competitions as a standardized format.

Beyond increasing trust in model checking, certificates can be utilized in many other scenarios. For instance, such certificates will allow the use of model checkers as additional hammers in interactive theorem provers such as Isabelle [273] via Sledgehammer [274], with the potential of significantly reducing the effort needed for using theorem provers in domains where model checking is essential, such as formal hardware verification, our main application of interest. Currently in Isabelle, Sledgehammer allows to encode the current goal for automatic theorem provers or SMT solvers and then call one of many tools to solve the problem. The tool then provides a certificate which is lifted to a proof that can be replayed in Isabelle. We plan to add our model checker as an additional hammer to increase the automatic proof capability of Isabelle. This further motivates us to investigate certificate trimming via SAT proofs.

9.9 Appendix

This Appendix first provides two observations relating witness circuits under the new restricted simulation relation to the stratified simulation relation introduced in previous work [12]. The next section provides formal proofs for the correctness of Theorem 9.10 and thereby the formal basis of our certification approach based on restricted simulation. The final section formally proves the completeness of the witness circuit construction for phase abstraction.

9.9.1 Comparison: Stratified and Restricted Simulation

Previous work [12] introduced a more restricted form of simulation relation. The following two propositions show that the new format subsumes this work, and all previously introduced techniques for witness circuit construction can still be applied.

Proposition 9.18. *Given circuits $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$, both stratified. If C' simulates C under the stratified simulation relation [13], then C' also simulates C under the restricted simulation.*

Proposition 9.19. *Given a circuit $C = (I, L, R, F, P)$ and its stratified k-witness circuit $C' = (I', L', R', F', P')$ as defined in [12]. Then C' is a witness circuit of C according to Definition 9.11.*

9.9.2 Correctness: Restricted-Simulation-Based Witnesses

The main claim for the correctness of our certification approach is given in Theorem 9.10. The formal proof of this theorem is split over the following lemmas relating to the reset, transition and property aspects of Def. 9.7. The first of these lemmas is presented in the main paper, and restated here for completeness.

Lemma 9.20. *Let $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ be two stratified circuits satisfying the reset condition defined in Def. 9.7.1. Then $R'(L \cap L')$ is semantically dependent only on their common variables.*

Proof. We provide a proof by contradiction. Given a latch $l_a \in (L' \setminus L)$ and a latch $l_b \in (L \cap L')$, suppose that $r'_{l_b}(I', L')|_{l_a} \not\equiv r'_{l_b}(I', L')|_{\neg l_a}$ which entails that $r'_{l_b}(I', L')$ is dependent on l_a. Since $l_a \in (L' \setminus L)$, we have $l_a \notin L$. Based on this, we have $r_{l_b}(I, L)|_{l_a} \equiv r_{l_b}(I, L)|_{\neg l_a}$ since l_a is not a variable in C. Let s be a satisfying assignment to $r_{l_b}(I, L)|_{l_a}$ as well as $r_{l_b}(I, L)|_{\neg l_a}$. By the reset condition $r_{l_b}(I, L) \equiv r'_{l_b}(I', L')$, the same assignment s satisfies $r'_{l_b}(I', L')|_{l_a}$ and $r'_{l_b}(I', L')|_{\neg l_a}$. Then we have $r'_{l_b}(I', L')|_{l_a} \equiv r'_{l_b}(I', L')|_{\neg l_a}$, which contradicts the fact that $r'_{l_b}(I', L')|_{l_a} \not\equiv r'_{l_b}(I', L')|_{\neg l_a}$. The same argument can be applied to the inputs. Therefore, every $r'_l(I', L')$ for $l \in (L \cap L')$ is dependent only on common variables $(L \cap L') \cup (I \cap I')$. Since $R'(L \cap L')$ is defined as $\bigwedge_{l \in (L \cap L')} l \simeq r'_l(I', L')$, the same follows. □

Lemma 9.21. *Given two stratified circuits $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ that satisfy Def. 9.7.1. For every reset state in C, there is a reset state in C' with the same assignments to $L \cap L'$.*

Proof. Let s be a satisfying assignment to $R(L)$. Under the reset condition $r_l(I, L) \equiv r'_l(I', L')$ for $l \in (L \cap L')$, the same assignment satisfies $R'(L \cap L')$. Since the reset functions of $L \cap L'$ can be assumed to not have dependencies on other latches $L' \setminus L$ according to Lemma 9.9, and the rest of $R'(L')$ (i.e., $R'(L' \setminus L)$) is also stratified, we can assume a topological ordering of the dependency graph where all latches in $L' \setminus L$ come before $L \cap L'$. We can therefore assign values to the rest of the latches $L' \setminus L$ by traversing in reverse order. □

That is a reset state in C can always be extended to a reset state in C' and similar arguments can be applied to transitions from current to next state:

Lemma 9.22. *Given two stratified circuits $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ that satisfy the transition condition (Def. 9.7). Then $f'_l(I', L')$ for all $l \in (L \cap L')$ semantically only depend on latches in $L \cap L'$.*

Proof. The proof follows the same logic as that of Lemma 9.9. □

Lemma 9.23. *Given two stratified circuits $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ that satisfy the transition condition (Def. 9.7.2). Let u be a state in C and u' a state in C' such that the common variables are assigned to the same values in both u and u'. Then for every successor of u, there is a successor state of u' with the same assignments to $L \cap L'$.*

Proof. First of all, let s be a satisfying assignment to U_1 (i.e., $L_1 \simeq F(I_0, L_0)$). We construct a satisfying assignment s' to U'_1 by first applying the same variable assignment to $L_0 \cap L'_0$ in C' (as well as $I_0 \cap I'_0$) and extend it to an assignment of $I'_0 \cup L'_0$. Since $l_1 \simeq f_l(I_0, L_0)$ for all $l \in L_0$, and the transition condition holds, by Def. 2, $l_1 \simeq f'_l(I'_0, L'_0)$ for all $l \in (L \cap L')$. Therefore, together with Lemma 9.22, we can keep the same variable assignment to $L_1 \cap L'_1$ in s'. The rest of the variables $L'_1 \setminus L_1$ have their transition functions solely depend on variables from I'_0 and L'_0 therefore s' is guaranteed to be satisfiable.

□

Theorem 9.24. *Let $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ be two stratified circuits, where C' simulates C under restricted simulation.*
If C' is safe, then C is also safe.

Proof. We assume C' is safe, and provide a proof by contradiction by assuming C is not safe. Suppose there is an assignment s over $I \cup L$ satisfying $R(L_0) \wedge U_m \wedge \neg P(I_m, L_m)$. By Lemma 9.21, the same assignment of $R(L_0 \cap L'_0)$ can be extended to satisfy $R'(L'_0)$. Furthermore, by Lemma 9.23, we can construct a satisfying assignment for $R'(L'_0) \wedge U'_m$ such that the common latches are always assigned the same values as according to s. Thus, by assumption that C' is safe, $P'(I'_m, L'_m)$ follows which together with Def. 9.7.3 results in a contradiction. □

9.9.3 Completeness: Phase Abstraction Witness Construction

The presented witness circuit construction for phase abstraction will always produce a valid witness circuit. This claim is presented for each step of the construction in Theorem 9.12, 9.15, and 9.17. The proofs are presented in the following. For Theorem 9.15, the definition of unfolded loop invariant is restated in greater detail.

Theorem 9.25. *Given a circuit $C = (I, L, R, F, P)$ and its reduced circuit $C' = (I', L', R', F', P')$. A witness circuit of C' is also a witness circuit of C.*

Proof. Let $W' = (J', M', S', G', Q')$ be a witness circuit of C'. First of all, we show that W' simulates C. As $L' \subseteq L, K = L' \cap M'$ is also the common set of latches between C and W'. The resets of C remain stratified. As the reset functions and transition functions of K stay the same, together with Def. 9.6 where $P' = P$, the three checks in Def. 9.7 are satisfied. We conclude W' simulates C. By Def. 9.11, Q' is an inductive invariant. Thus W' is also a witness circuit of C. $\square$

Definition 9.26 Unfolded cube lasso**.** Given a circuit $C = (I, L, R, F, P)$ with a phase number $n \in \mathbb{N}^+$, and its unfolded circuit $C' = (I', L', R', F', P')$.
Let $c_0, \cdots, c_d, \cdots, c_\delta, \cdots, c_{\delta+\omega}$ be the cube lasso of C. The unfolded cube lasso $c'_0, \cdots, c'_{\delta'}, \cdots, c'_{\delta'+\omega'}$ is defined as follows: (i) $\delta' * n + d = \delta; \omega' * n + n - 1 = \omega$. (ii) For $i \in [0, \delta' + \omega'), c'_i = \bigwedge\limits_{j \in [0,n)} c_{i*n+j+d}(I^j_i, L^j_i)$.

Definition 9.27 Unfolded loop invariant**.** Given a circuit $C = (I, L, R, F, P)$ and its unfolded circuit $C' = (I', L', R', F', P')$, and a cube lasso $c_0, \cdots, c_m$ of C. Let $c'_0, \ldots, c'_{m'}$ be the unfolded cube lasso.
The unfolded loop invariant ϕ is defined as $\bigvee\limits_{i \in [0,m']} c'_i$.

Lemma 9.28. *The unfolded cube lasso is a cube lasso in the unfolded circuit.*

Lemma 9.29. *Given a circuit $C = (I, L, R, F, P)$ and its unfolded circuit $C' = (I', L, R', F', P')$ with a phase number n. Let $\Gamma \subseteq L$ be a set of latches that are associated with periodic signals determined from a cube lasso $c_0, \ldots, c_{\delta+\omega}$ of C. The unfolded loop invariant ϕ is an inductive invariant in the circuit C' for the property $\bigwedge\limits_{l \in \Gamma} (\bigwedge\limits_{i \in [0,n]} (l^i \simeq \lambda^i_l))$, where l^i is a temporal copy of l.*

In the above lemma, the property states in essence that the periodic signals always have the values according to their periodic patterns in the unfolded cube lasso. Each cube in the unfolded cube lasso is a partial assignment to L', which consists of n copies of the original set of latches L. Therefore we use temporal copies such as λ^i_l to represent the periodic values of the latches.

Theorem 9.30. *Given circuit $C = (I, L, R, F, P)$, and factor circuit $C' = (I', L', R', F', P')$. Let $W' = (J', M', S', G', Q')$ be a witness circuit of C', and $W = (J, M, S, G, Q)$ constructed as in Def. 9.14. Then W is a witness circuit of C.*

Proof. First of all we show the composite witness W simulates C. W' and C are stratified and R only references latches in L_1 thus no new cyclic dependencies is introduced. Therefore W is stratified too. L is the common set of latches for W and C. By Def. 9.4, the reset and transition functions of L are the same in both circuits, this satisfies the reset check as well as the transition check. Since W' is a witness circuit of C', we have $Q' \Rightarrow P'$ and therefore $Q \Rightarrow P'$. We omit the rest of the proof as it follows the same logic as Theorem 2 in [13]. □

Theorem 9.31. *Given a circuit $C = (I, L, R, F, P)$ with a phase number $n \in \mathbb{N}^+$, its unfolded cicuit $C' = (I', L', R', F', P')$ with a witness circuit $W' = (J', M', S', G', Q')$. Let $W = (J, M, S, G, Q)$ be the circuit constructed as in Def. 9.16. Then W is a witness circuit of C.*

Proof. First, we prove the simulation relation. The L^0 latches have the same reset functions as in C, thus they are stratified, and they do not have dependencies on other latches outside L^0. The resets of N are the same as in W' thus also stratified. Based on Def 9.16, the rest of the latches do not depend on other latches therefore S is stratified. Let $K = L \cap M$. By Def. 9.16, for $l \in L^0$, $s_l \equiv r'_l$. Together with Def. 9.3, we have $R(K) \Rightarrow S(K)$. By Def. 9.16, $f_l \equiv g_l$ for $l \in K$. Also by q^0, we have $Q \Rightarrow P$. Therefore, W simulates C.

Next we show that the BMC check passes for W such that $S(M) \Rightarrow Q(J, M)$. Since only b^0 is set at reset, q^1, q^2, q^3 and q^5 are satisfied. All e^is are set to $\perp$ at reset, thus q^6, q^7, q^8 are also satisfied. The reset in Def. 9.16 directly implies $R'(L^0)$ and $S'(N)$, and by Def. 9.3 it also satisfies $R(L^0)$, which satisfies q^4. Based on Def. 9.7, $R'(L^0) \Rightarrow S'(M' \cap L^0)$, which together with the stratification of S', results in $S'(M')$. This implies $Q'(J', M')$, based on the BMC check of the inductive invariant Q'. By Def. 9.7, we have $P'(I', L')$, and based on Def. 9.3 this gives us $P(I^0, L^0)$ thus q^0 is also satisfied.

Let V_1 be the unrolling of W and we move on to prove $V_1 \wedge Q(J_0, M_0) \Rightarrow Q(J_1, M_1)$ by providing a proof by contradiction. We assume $V_1 \wedge Q(J_0, M_0) \wedge \neg Q(J_1, M_1)$ has a satisfying assignment, and fix this assignment. We consider two cases based on the values of b^i_0: (i) all b^i_0 are set to $\top$; (ii) B_0 is partially initialised, i.e., not all b^i_0 set to $\top$. We begin with the first case. Since b^is always transition to $\top$ or b^{i-1}, and under the assumption that all b^is are $\top$, q^1_1, q^2_1, q^4_1 and q^7_1 are immediately satisfied. By Def. 9.16, L^0 transitions in the same way as in C, and the rest of the latches are simply copying the values during the transition, thus q^3_1 is satisfied. We move on to consider q^5_1. Based on q^5_0, we get a satisfying assignment for $\bigvee_{i\in[0,n)} ((\bigwedge_{j\in[i,n-1)} \neg e^j) \wedge (\bigwedge_{j\in[0,i)} e^j) \wedge Q'(J^0, M' \cap (L^i \cdots \cup L^{i+n-1} \cup N))$. We thus consider three different cases based on the disjunction and the value of e^i_0s.

- Case where all e^i_0s are $\perp$: based on Def. 9.16, e^0_1 is set to $\top$ and the rest set to $\perp$, thus q^6_1, q^8_1 are satisfied. Since e^{n-2}_0 is set to $\perp$, the latches of N stay the same after the transition. In this case, we already have $Q'(J^0_0, M'_0 \cap (L^0_0 \cdots \cup L^{n-1}_0 \cup N_0))$ satisfied, together with the transition function defined and q^3_0, the same assignment satisfies $(\bigwedge_{j\in[1,n-1)} \neg e_1^j) \wedge e_1^0 \wedge Q'(J_1^0, M_1' \cap (L_1^1 \cup L_1^n \cup N_1))$, which satisfies q_1^5. By Def. 9.7, we have $P'(I_1', L_1')$ which also implies $P(I_1^0, L_1^0)$. Therefore q_1^0 is also satisfied.

- Case where not all e_0^i set to $\top$: let k be the index such that $0 < k < n-2$, $e_0^i \simeq \top$ for $i \in [0,k]$ and $e_0^i \simeq \bot$ for $i \in (k, n-2]$, and after the transition $e_1^i \simeq \top$ for $i \in [0, k+1]$ and $e_1^i \simeq \bot$ for $i \in [k+2, n-2]$, which immediately satisfies q_1^6 and q_1^8. Based on q_0^5, we already have $Q'(J_0^0, M_0' \cap (L_0^{k+1} \cdots \cup L_0^{k+n} \cup N_0))$. Similarly as the case above, e_0^{n-2} is $\bot$ thus the rest follows.

- Case where all e_0^is are $\top$: based on Def. 9.16, all e_1^is become $\bot$, thus q_1^6 and q_1^8 satisfied. As stated in Def. 9.16, when $e_0 \equiv \top$, latches in N transition as in W'. In this case, based on q_0^5, we already have $Q'(J_0^1, M_0' \cap (L_0^{n-1} \cdots \cup L_0^{2n-2} \cup N_0))$ satisfied, together with the transition function defined and q_0^3, the same assignment satisfies $Q'(J_1^0, M_1' \cap (L_1^0 \cup L_1^{n-1} \cup N_1))$, which satisfies q_1^5. By Def. 9.7, we have $P'(I_1', L_1')$ which implies $P(I_1^0, L_1^0)$. Therefore p_1^0 is satisfied.

In either case, we can apply the same assignment to satisfy $Q(J_1, M_1)$ and therefore reach a contradiction. The rest of the proof follows similar logic. □

Chapter 10

Certifying Constraints in Hardware Model Checking

Submitted

Authors Nils Froleyks, Emily Yu, Armin Biere, and Keijo Heljanko

Changes from Published Version Corrected typos and adjusted layout.

Authors Contributions The author contributed to the certificate constructions detailed in the paper and provided their completeness proofs. The extraction techniques were collected and formalized by E. Yu. The weaker certificate checks covering constraints, as presented in this paper, resulted from discussions with E. Yu and were motivated by a GitHub issue raised by Andrew Luka. The weak induction check was devised by the author. All presented tools and experiments were developed by the author.

Acknowledgement This work is supported in part by the ERC-2020-AdG 101020093, the LIT AI Lab funded by the State of Upper Austria, the Research Council of Finland under the project 336092, and a gift from Intel Corporation.

Furthermore we of course also owe a big thank-you to the submitters of model checkers and benchmarks to the competition over all these years. Without their enthusiasm and support neither the competition nor this study would exist.

Abstract Model checking is a powerful automated reasoning technique for verifying hardware designs, ensuring that they function correctly before deployment. However, modern model checkers are complex software systems with hundreds of thousands of lines of code, making them prone to errors. To increase confidence in verification results, recent efforts in hardware verification focus on requiring model checkers to produce machine-checkable proofs according to a standardized format that can be independently validated. Yet, implementing proof generation across different verification algorithms presents a unique challenge.

One critical aspect of hardware model checking is handling constraints—assumptions about the system's environment that help simplify analysis, improve performance, and extend the applicability of verification techniques. Like most industrial verification languages, the AIGER format, used in hardware model checking competitions, explicitly supports constraints. This paper addresses the challenge by developing a certification approach that ensures verification results remain trustworthy when constraints are present. We introduce certificate generation methods for three classes of constraints that can be extracted from the models. Furthermore, to support a broader range of constraints and more complex reset logic for industrial use, we also provide alternative Quantified Boolean Formula checks for a standardized proof format with a single quantifier alternation. Additionally, we extend this framework by incorporating k-induction with uniqueness constraints. We implement these techniques in a certification toolkit, and provide empirical evaluation on competition benchmarks, demonstrating their effectiveness.

10.1 Introduction

Model checking is a widely used technique in formal verification to ensure that hardware designs, such as processors or circuits, operate correctly before they are manufactured. This automated process checks whether a design in the form of a gate-level netlist satisfies specific requirements—often expressed as logical properties—by exploring all possible executions of the system. However, the automated reasoning tools that perform model checking, known as model checkers, are complex, consisting of hundreds of thousands of lines of code. This complexity introduces the risk of errors within the tools themselves, raising questions about the reliability of their results. To address this, recent developments in hardware verification emphasize the generation of certificates—formal proofs that can be independently verified by another program to confirm the verdicts of the model checkers [237, 243, 246, 253, 275].

As in most industrial verification languages [276, 277], following the assume-guarantee methodology [278], an essential feature of hardware model checking is the use of *constraints*, which represent assumptions about the environment in which the hardware operates. These assumptions simplify the verification task, improve computational efficiency, and allow model checking to tackle more complex designs. The AIGER format [51], a standard in hardware model checking competitions (HWMCCs) [21], explicitly incorporates constraints as part of its benchmark specifications. In the latest iteration, HWMCC'24, certification became mandatory: all participating model checkers had to produce a machine-checkable certificate when confirming a design's correctness. The verification pipeline is illustrated in Fig. 10.1.

The standardized certificate format [11], in the form of a witness circuit, relies on five simple satisfiability (SAT) checks and a polynomial-time check for reset function acyclicity (called stratification). This approach is efficient and compatible with many model checking algorithms. However, challenges arise when designs include complex reset logic–such as cyclic dependencies—which complicates certificate generation, but are common in industrial practice [279–281]. To overcome this, we propose alternative

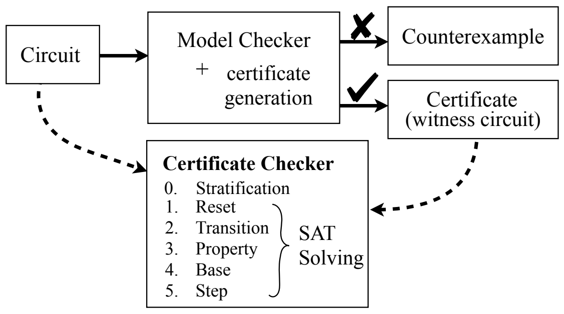

Figure 10.1: Overview of the model checking certification process. The generated certificates are independently validated by a certificate checker using five simple SAT checks and a polynomial-time check on stratification.

checks using Quantified Boolean Formula (QBF) in the same format. This relaxed format is easily adapted by the certificate checker CERTIFAIGER used in HWMCC'24, offers more flexibility for model checker developers while maintaining trust in the results.

Since commercial designs often require hundreds of constraints to accurately model their environments, verification tools must go beyond the explicit constraints provided in the design description. *Hidden constraints* can often be inferred from the model itself [282, 283]. In fact, such constraint extraction is also implemented in the state-of-the-art model checker ABC [177], as documented in its official manual. Complementary efforts have also focused on the efficient synthesis of such constraints [284, 285]. These constraints, identified through preprocessing, refine the verification problem by focusing only on parts of system states, preserving the validity of the original property. While this simplification enhances efficiency, the resulting certificate from the model checker applies only to the reduced model, not the original. This therefore adds difficulty for certification.

Our contribution. This paper addresses this gap by demonstrating how to generate a certificate that validates the original design, accounting for the extracted constraints. Furthermore, we consider k-induction [35], a powerful model checking method that proves properties by examining inductive behaviours. This technique often employs uniqueness constraints (sometimes also called simple path constraints) to ensure that states in the induction step do not repeat, making it a complete verification approach. Certifying k-induction with such constraints has been challenging [12], but we present a solution using QBF-based checks as well as a method to generate corresponding certificates.

We summarize our main contributions as follows.

1. *Certification of extracted constraints.* We introduce a method for generating certificates that account for three classes of extracted constraints: *model constraints*, *inductive constraints*, and *property constraints*. The resulting certificates validate that the verification result holds for the original model, prior to preprocessing.

2. *Relaxed certificate format.* To accommodate more complex encoding logic in modern model checking techniques, we propose alternative QBF-based certificate checks. These relaxations extend the standardized certificate format used in hardware model checking competitions and remain verifiable by an independent checker.

3. *Certification of uniqueness constraints in k-induction.* We present a generalized certification approach by extending the certificate format with the notion of oracles, enabling certification of uniqueness constraints.

4. *Empirical evaluation.* We evaluate our approach on a set of HWMCC benchmarks, demonstrating its practicality and effectiveness.

10.2 Constrained circuits

In the rest of the paper, we employ the following notation: let $\mathcal{V}$ be a set of Boolean variables, we consider formulas over $\mathcal{V}$ with the Boolean operators $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$. The last denotes equivalence and can have an infix negation $\not\leftrightarrow$. A (partial) assignment gives each variable in (a subset of) $\mathcal{V}$ the value *true*($\top$) or *false*($\bot$). Applying a function f to an assignment s, denoted as $f(s)$, follows the usual semantics. If s is a total assignment, $f(s)$ yields a truth value; if s is partial, $f(s)$ results in a formula not dependent on any variables in s. By *dependent* we mean syntactically dependent, i.e. no variable v in s appears in $f(s)$. For a partial assignment, we use *extension* to refer to an assignment that assigns more variables and is otherwise the same. For sets of Boolean variables $\mathcal{U}$ and $\mathcal{V}$, we denote union as $\mathcal{U}, \mathcal{V}$ and as $\mathcal{U} \dot\cup \mathcal{V}$ when we want to emphasize that $\mathcal{U}$ and $\mathcal{V}$ are disjoint. For next-state variables, we write $\mathcal{V}_1$ to denote a copy of $\mathcal{V}$, and for symmetry we then refer to the original copy $\mathcal{V}$ as $\mathcal{V}_0$. We extend this notation to any number of transitions.

We consider hardware designs modeled as finite logical circuits [253]. Each circuit is associated with a safety property, and an environment constraint, encoded as Boolean formulas over input and latch variables. A state of the system is an assignment to inputs and latches satisfying the constraint. The values of the latches are initialized using their reset function and evolve according to the transition functions. At every timestamp, the values of latches are determined by the values of inputs and the previous latch values. The inputs can have arbitrary values and represent non-determinism.

Definition 10.1 Circuit. A circuit $M = (I, L, R, F, P, C)$ is defined as a tuple consisting the following attributes:

1. I: a finite ordered set of Boolean input variables;

2. L: a finite ordered set of Boolean latch variables;

3. $R = \{r_l(I, L) \mid l \in L\}$, where $r_l(I, L)$ is a reset function associated with a latch l;

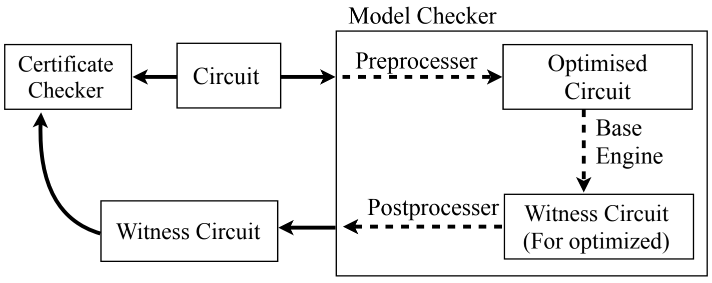

Figure 10.2: A compositional approach for the certification flow when preprocessing is employed in a model checker.

4. $F = \{f_l(I, L) \mid l \in L\}$, where $f_l(I, L)$ is a transition function associated with a latch l;
5. $P(I, L)$ represents the set of good states; and
6. $C(I, L)$ encodes the set of constrained states.

In this paper, we focus on safety properties, which holds in all good states. For circuits that do not come with explicitly given constraints, we simply consider $C(I, L) = \top$. When there are multiple constraints $c_0, c_1, ...$, we take their conjunction to form a single one $C(I, L) = \bigwedge c_i$. A circuit is said to be *safe* if the property P holds in all constrained states reachable from the initial states. The set of initial states are defined by $R\{L\} = \bigwedge_{l \in L}(l \leftrightarrow r_l(I, L))$. The same notation is used for transitions $F_{0,1}\{L\} = \bigwedge_{\ell \in L}(\ell_1 \leftrightarrow f_\ell(I_0, L_0))$. Both are also used for subsets of L. The convention to use indices on formulas–while omitting explicit variable references–extends to other circuit components, e.g., C_0 stands for $C(I_0, L_0)$.

When designing a certificate format, one of the key objectives is to eliminate the need for quantifiers in certificate checking, allowing the process to fall within the co-NP complexity class. For this purpose, the authors of [12] introduced the concept of *stratified* reset functions, i.e., the reset functions of a circuit do not have cyclic dependencies. This entails that the formula $R\{L\}$ is always satisfiable.

A circuit $M = (I, L, R, F, P, C)$ is safe, if P holds in all states reachable under the constraint. A counterexample is a path $s_0, s_1, \ldots, s_n$ from a reset state s_0 to a bad state s_n where all s_i satisfy the constraint:

$$R_0\{L\} \;\wedge \bigwedge_{i\in[0,n)} F_{i,i+1}\{L\} \;\wedge \bigwedge_{i\in[0,n]} C_i \;\wedge\; \neg P_n.$$

Example 10.2. A common problem in hardware design is to arbitrate the usage of shared resources. In the following, we illustrate with such an example, also described in Fig 10.3. Each user i can send a request for the shared resources setting the input signal req_i and gains access from the acknowledge signal ack_i. The crucial property is that no two users get acknowledged at the same time. The internal design of the arbiter can be

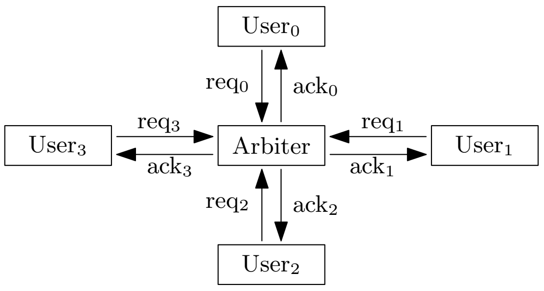

Figure 10.3: An arbiter for four users.

quite complex when additional properties, such as fairness, are considered. To make the model checking task easier, we specify the constraint as $C = \bigwedge_{i \neq j} \neg(ack_i \wedge ack_j)$. This constraint can either be extracted by the model checker in a preprocessing step (we later discuss in Sec. 10.3), or it can be specified explicitly by the hardware designer. In both cases, the implementation details of the arbiter might be abstracted away from the design.

Counterexamples are used to validate model checking failures. Conversely, when model checking succeeds, we rely on a different certificate format–the witness circuit (described in [11])–which is also the standard format used in the hardware model checking competitions.

Definition 10.3 Witness Circuit. $W = (I', L', R', F', P', C')$ is a witness circuit of circuit $M = (I, L, R, F, P, C)$, both with constraints, if R' is stratified and for $K = L \cap L'$:

1. Reset: $R\{K\} \wedge C \rightarrow R'\{K\} \wedge C'$;
2. Transition: $F_{0,1}\{K\} \wedge C_0 \wedge C_1 \wedge C'_0 \rightarrow F'_{0,1}\{K\} \wedge C'_1$;
3. Property: $(C \wedge C') \rightarrow (P' \rightarrow P)$;
4. Base: $R'\{L'\} \wedge C' \rightarrow P'$;
5. Step: $P'_0 \wedge F'_{0,1}\{L'\} \wedge C'_0 \wedge C'_1 \rightarrow P'_1$.

Each condition defined above is encoded as a SAT formula. An additional polynomial-time check that R' is stratified [12] is needed, such that there is no cyclic dependencies among the reset functions.

10.3 Certifying Constraint Extraction

Constraint extraction is a powerful preprocessing step that simplifies verification for any base model checking engine [282, 283], such as k-induction. This is also one of the essential features implemented by ABC [177]. In this section, we address the challenge of certifying the hidden constraints uncovered by this process, as illustrated in Fig. 10.2. Because a witness circuit generated by the base engine only proves correctness on the preprocessed circuit, it does not immediately satisfy Def. 10.3. Consequently, postprocessing the witness circuit is necessary to produce a proper certificate for the original problem.

10.3.1 Model Constraints

The first type of constraints we consider is *model constraints*. Intuitively, they are those that hold in all reachable states, similar to a safety property. It is therefore evident that adding them does not change the set of reachable states in the model. In industry practice, it is common to verify hundreds of properties for the same design, and it is beneficial for the verification performance to add already proven properties as constraints [286, 287].

Definition 10.4 Model constraint**.** Given a circuit $M = (I, L, R, F, P, C)$, the formula $D(I, L)$ is said to be a model constraint if it holds in all reachable states, i.e., the following formula is unsatisfiable for any n:

$$R_0\{L\} \;\wedge \bigwedge_{i\in[0,n)} F_{i,i+1}\{L\} \;\wedge \bigwedge_{i\in[0,n]} C_i \;\wedge\; \neg D_n.$$

By Def. 10.4, during model checking, the extracted model constraints can simply be appended to strengthen C to form a new constraint $C \wedge D$. A base model checker that uses for instance IC3/PDR [34] or k-induction can be then employed. Once verification is completed, the base engine also provides a witness circuit. Next, we are going to show how to postprocess such witness circuit to obtain one that certifies the safety of the original circuit. For that we are going to compose it with another witness circuits for the model constraint D.

We assume inputs and latches not shared with the model to be unique to the respective witness circuits. Otherwise renaming is necessary.

Theorem 10.5. *Given model $M = (I, L, R, F, P, C)$ with constraint D. Let $M_P = (I, L, R, F, P, C \wedge D)$ be the model under the addition of constraint D, with witness $W_P = (I^P, L^P, R^P, F^P, P^P, C^P)$. Further let $W_D = (I^D, L^D, R^D, F^D, D^D, C^D)$ be the witness certifying that D is a model constraint, i.e, a witness for the circuit $M_D = (I, L, R, F, D, C)$. Assuming the inputs and latches of W_P and W_D to be disjunct with the exception of those also shared with M, the circuit $W = (I^P \cup I^D, L^P \cup L^D, R^P \cup R^D, F^P \cup F^D, P^P \wedge P^D, C^P \wedge C^D \wedge D)$ is a witness for M.*

Proof. Before we show that W passes every check of Def. 10.3, we note that W_P and W_D not sharing any additional inputs or latches implies that $R^P \cup R^D$ is still stratified. Further, given the reset check for both W_P and W_D the reset states in W are simply: $(R^P \cup R^D)\{L^P \cup L^D\} = R^P\{L^P\} \wedge R^D\{L^D\}$. The same is true for encoding transitions. We begin with the reset check and show that the premise implies D by the witness relation between M_D and W_D:

$$R\{K\} \wedge C \Rightarrow R^D\{K\} \wedge C^D \Rightarrow R^D\{L^D\} \wedge C^D \Rightarrow P^D \Rightarrow D$$

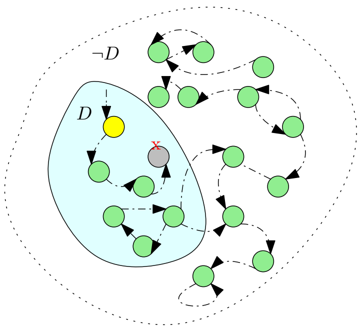

Figure 10.4: An illustration of the state space for inductive constraints. The bad states are marked gray; and the initial state is marked yellow. The blue region are the states satisfying the constraint D. Note that D is of course not a subset of $\neg D$.

Now the premise for the reset check for both W_P and W_D are met, and the conclusion of the reset check for W follows. The premise of the transition check of W gives us D_0, which by the transition check of W_D gives $F^D \wedge C_1^D$ and further D_1 using the step check. Again the premises for the step check of both W_P and W_D are fulfilled. For the remaining three checks, no additional arguments are needed as the conclusions are the same as in the witness checks for W_P and W_D and the premises have only been strengthened. □

Another interesting application of the described construction is the efficient certification of multi-property circuits. Consider a model $M = (I, L, R, F, P, C)$, where $P = \bigwedge_{i \in [0,n]} P^i$, and each P^i typically concerns only a sub-circuit of the original. A model checker may verify these properties individually and later compose the corresponding witness circuits using the same construction as above. In this case both witnesses are produced without the addition of an explicit model constraint and the D in the constraint definition of W is simply true. Any number of witness circuits may be composed through repeated application of the construction, which corresponds to taking a union over all witnesses at each component.

10.3.2 Inductive Constraints

Similar to model constraints, there is the notion of inductive constraints [282], which after extraction from the model can be used to strengthen the already present constraint. This is also partially motivated by inductive constraints in backward reachability [167].

Definition 10.6 Inductive constraint**.** An inductive constraint $D(I, L)$ of a circuit $M = (I, L, R, F, P, C)$ satisfies:
(1) $F_{0,1}\{L\} \wedge \neg D_0 \to \neg D_1$; and (2) $\neg D \to P$.

We illustrate the nature of inductive constraints in Fig. 10.4. Intuitively, it is correct to add any inductive constraint, since no bad traces can be eliminated. Once a trace leaves D it cannot cross back, since $\neg D$ is inductive, and no bad state can be reached with $\neg D \to P$. Therefore the model checker only needs to ensure P holds in the states that satisfy D (i.e., the area marked blue). Note the slightly confusing naming of *inductive* constraints taken from [282]; it is the negation of D that is inductive, and moreover it is not an inductive invariant, as it does not necessarily hold in the reset states.

We now show how to produce witness circuits for models with added inductive constraints. The following theorem requires the constraint to also be inductive with respect to the transition function of the witness under constraint W_D. This does not impose a practical limitation, since D only depends on inputs and latches from the original circuit and witness circuits typically do not introduce additional transitions for original latches. If this does happen to be the case, F' can be restricted to the original transitions without changing the success of the transition or step check in Def. 10.3.

To simplify the presentation, we denote a modified set of reset and transition functions like this $R' \vee \neg D$ and $F' \vee \neg D_1$. This construction can be easily achieved by introducing a fresh input i_l for each latch l, and encoding the formula D over these inputs to obtain D_I. With this setup, the new reset function for each latch l is defined as: $\texttt{ite}(D_I, r'_l, i_l)$, which selects r'_l when D_I holds, and otherwise takes the value of i_l. The transition function $\texttt{ite}(D_I, f'_l, i_l)$, similarly allows to transition to any state violating D.

Since R' is stratified and D_I depends only on inputs, the resulting reset function remains stratified.

Theorem 10.7. *Given model $M = (I, L, R, F, P, C)$, the constrained model $M_D = (I, L, R, F, P, C \wedge D)$ with witness $W_D = (I', L', R', F', P', C')$, where D is an inductive constraint in M and $F' \wedge \neg D_0 \rightarrow \neg D_1$, the unconstrained witness $W = (I', L', R' \vee \neg D, F' \vee \neg D_1, P' \vee \neg D, C' \vee \neg D)$ is a witness for M.*

Proof. We consider an arbitrary assignment $L_0 \cup L'_0 \cup I \cup I' \cup L_1 \cup L'_1$. If it satisfies $\neg D$ then also P and the property, reset and base check are trivially fulfilled. Similarly, with the assumption $\neg D_1$ the transition and step check hold. If D holds, any violation of the property, reset or base check would also be a violation for the witness relation of M_D and W_D. Since $\neg D$ is inductive under F and F', D_1 implies D_0 if the assignment violates either the transition or step check. In both cases, it would also contradict the witness relation of M_D and W_D. □

10.3.3 Property Constraints

We now take a look at another class of constraints, namely property constraints, that can be generated from a model by a dedicated constraint mining algorithm. Intuitively, property constraints are implied by the property itself.

Definition 10.8 Property constraint**.** A property constraint $D(I, L)$ of a circuit $M = (I, L, R, F, P, C)$ satisfies: $P \rightarrow D$.

Once property constraints are extracted, their use differs from the previous two constraint types, as both model constraints and inductive constraints are directly appended to the explicit constraint i.e., $C \wedge D$. Property constraints, on the other hand, require to be integrated into the transition functions.

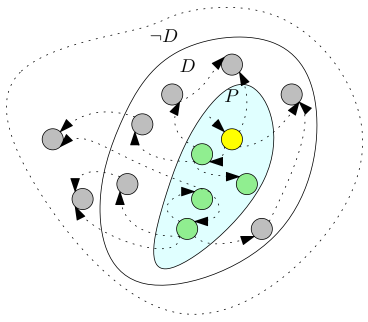

Figure 10.5: An illustration of property constraints. Since the property constraints act as conditions in the transition function, outgoing transitions only take place in states that satisfy D. There is no transition from states in $\neg D$ to D.

Model checking under property constraints. Given a circuit $M = (I, L, R, F, P, C)$ with property constraint D. The circuit under the property constraint is $M_D = (I, L, R, F_D, P, C)$ where: $F_D = \{\texttt{ite}(D_0, f_l, l) \mid l \in L\}$.

This suggests that at every step, if the property constraint holds in the current state, then it follows the same transition as before, otherwise it stays in the current state. We illustrate in Fig. 10.5 that adding property constraints does not change the model checking result. In other words, if the circuit under property constraints is proven to be safe, then this implies the original circuit is also safe. This can be validated by a certificate, constructed as follows.

Theorem 10.9. *Given model $M = (I, L, R, F, P, C)$, the model under property constraint $M_D = (I, L, R, F_D, P, C)$ with witness $W_D = (I', L', R', F'_D, P', C'_D)$, the witness circuit for property constraints $W' = (I', L', R', F', P', C')$, where $F' = \{ite(D_0, f'^D_l, f_l \mid l \in K\} \cup \{f'^D_l \mid l \in L' \setminus K\}$ and $C' = C'_D \wedge D$, is a witness for M.*

Proof. Given the witness relation between M_D and W_D, we need to proof that W is a witness for M. First we conclude that D holds in any reset of M valid under the constraint:

$$R\{K\} \wedge C \Rightarrow R'\{K'\} \wedge C'_D \Rightarrow R'\{L'\} \wedge C'_D \Rightarrow P' \Rightarrow P \Rightarrow D$$

Now, we assume an assignment s to $L_0 \cup L'_0 \cup I \cup I' \cup L_1 \cup L'_1$ violating any of the witness checks outlined in Def. 10.3. If D_0 does not hold, the reset check is fulfilled by the above argument. The other checks are trivially fulfilled as C'_0 is false. If D_0 holds in s, the assignment will also violate the same check for the witness relation between M_D and W_D. □

10.4 Relaxed simulation

According to Def. 10.3, in order for the Reset and Transition checks to pass, the constraint C' needs to hold for all possible extensions generated by the reset and transition functions. One way to check if this is the case, is to ensure that the constraint only contains variables shared between the two circuits, which is a natural assumption. For many model checking algorithms, for example IC3/PDR and BDDs under constraints, it is often the case that the certificate W is simply $W = (I, L, R, F, P', C)$ [288], where P' is the inductive strengthening generated by the algorithm itself, encoding an over-approximation of the reachable states, with the rest of the circuit unchanged from M.

However, for other algorithms where it is necessary to constrain new variables in the witness circuit as well as to allow cyclic reset definitions, the format in Def. 10.3 has to be relaxed to include quantifiers in the checks.

Definition 10.10 Quantified Reset and Transition**.** Consider the given circuit $M = (I, L, R, F, P, C)$, then the circuit W with $W = (I', L', R', F', P', C')$ is a witness circuit for the local witness variables $X' = (I' \cup L') \setminus (I \cup L)$ if it satisfies Def. 10.3, with the *Reset* and *Transition* conditions relaxed as follows:

- Reset$^\exists$: $R\{L\} \wedge C \rightarrow \exists X' \, [R'\{L'\} \wedge C']$;
- Transition$^\exists$: $F_{0,1}\{L\} \wedge C_0 \wedge C_1 \wedge C'_0 \rightarrow \exists X'_1 \, [F'_{0,1}\{L'\} \wedge C'_1]$.

If Reset$^\exists$ is not used, R' has to be stratified.

If the reset functions of W are cyclic, i.e., not stratified, it is necessary to use the reset$^\exists$ check. Without this existential quantification, it is not sufficient to prove the soundness theorem of Def. 10.3: *if W is a witness circuit of M, then M is safe*. Proving this theorem requires establishing a simulation relation between M and W, which in turn requires every reset state in M to correspond to a reset state in W.

Because the Reset$^\exists$ condition directly guarantees that any reset state in M aligns with a reset state in W, we no longer need to impose the stratification requirement on W. In other words, if Reset$^\exists$ is used, the stratification check becomes unnecessary. However, even with stratified resets, due to the presence of constraints, Reset$^\exists$ may still be required to ensure $R'\{L'\} \wedge C'$ remains satisfiable, thereby guaranteeing at least one valid reset state under the constraint in W.

Nevertheless, if the quantifier-free Reset check passes, certification remains valid, and the same holds for the Transition check. Furthermore, if the witness constraint does not use any variables in X' that are unique for the witness circuit, there is no need to employ the relaxed versions. All other checks remain unchanged from Def. 10.3. In the following, we formally prove that this format constitutes a correct certificate.

Theorem 10.11. *Given a circuit M and its witness circuit W that satisfies Def 10.10, then M is safe.*

Proof. We assume that W is a witness circuit for M, and do a proof by contradiction. Suppose W is not safe. Then there is a bad trace of some finite length n in the form of an assignment to $n + 1$ copies of $I \cup L$ satisfying:

$$R_0\{L\} \wedge C_0 \wedge F_{0,1}\{L\} \wedge C_1 \dots C_{n-1} \wedge F_{n-1,n}\{L\} \wedge C_n \wedge \neg P_n.$$

We extend this assignment to each copy of the variables in X' that satisfies:

$$R'_0\{L'\} \wedge C'_0 \wedge F'_{0,1}\{L'\} \wedge C'_1 \dots C'_{n-1} \wedge F'_{n-1,n}\{L'\} \wedge C'_n \wedge \neg P'_n.$$

The reset$^{\exists}$ and transition$^{\exists}$ checks directly imply the existence of a reset state in W, respectively a successor state. Since the transition$^{\exists}$ check only quantifies over the next state version of the extension variables X'_1 the trace in W can be constructed iteratively from reset to bad state. Applying the same argument n times yields an assignment to $(X')^n$ satisfying $F'_{i,i+1}\{L\}$ for $i \in [0, n)$ and C_i for $i \in [0, n]$. Lastly, the property check guarantees $\neg P'_n$, giving us the desired assignment. However, the Base and Step check together ensure that the property P' holds on all reachable states of W, thus contradicting the initial assumption that a bad trace exists in M. □

10.5 Extending Circuits with Oracles

The core mechanism to allow certification via a witness circuit is the addition of inputs and latches. It results in more behavior for which safety is easier to prove, i.e., following the classical concept of inductive strengthening. Although this works well for most model checking techniques, requiring inductiveness can be challenging for techniques which consider multiple execution paths at the same time. A prime example is k-induction with uniqueness constraints (Def. 10.16), where these constraints are justified by the fact that any reachable bad state can also be reached via a shorter loop-free path.

Even though certifying k-induction has been studied before [253, 289–291], none of these works address the use of *uniqueness constraints*. In this section, we extend witness circuits with *oracles*, enabling reasoning about multiple possible states within the same witness circuit. Intuitively, oracles—like inputs—can be assigned arbitrary values. However, we distinguish oracles from inputs so that they can be separately identified and correctly handled in the certificate checks.

Definition 10.12 Step check with oracles**.** Consider the circuit $M = (I, L, R, F, P, C)$, then the circuit $W = (I' \dot\cup O', L', R', F', P', C')$ where O' are the oracle inputs. W is a witness if R' and F' are independent of O' and the circuits satisfies the conditions in Def. 10.3, where the step check is relaxed to:

$$\text{Step}^{\exists} : \forall O'_0\, [P'_0 \wedge F'_{0,1}\{L'\} \wedge C'_0] \wedge C'_1 \;\rightarrow\; P'_1$$

If R' or F' are dependent on O', we need additional quantifiers in the respective checks similar to Def. 10.10.

Definition 10.13 Reset and Transition with oracles**.** Given a circuit M, the circuit W, with $M = (I, L, R, F, P, C)$, $W = (I' \dot\cup O', L', R', F', P', C')$, and $X' = (I' \cup L') \setminus (I \cup L)$, is a witness circuit if it meets Def. 10.3, where the step check has been relaxed to Step$^{\exists}$ from Def. 10.12 and either or both reset and transition checks have been relaxed to:

- Reset$^{\exists\forall}$: $R\{L\} \wedge C \;\rightarrow\; \exists X' \forall O'\, [R'\{L'\} \wedge C']$;

- Transition$^{\exists\forall}$: $F_{0,1}\{L\} \land C_0 \land C_1 \land C_0' \rightarrow \exists X_1' \forall O_1' \, [F_{0,1}'\{L'\} \land C_1']$.

The following we show the soundness of certificates with oracles.

Theorem 10.14. *A witness circuit that satisfies the certificate format with oracles is a valid certificate.*

Proof. Since the oracles are only in the witness circuit, the reset$^{\exists\forall}$ and transition$^{\exists\forall}$ conditions still allow to do the construction of the trace in W the same way as in the proof of Theorem 10.11. The same is true if R' or F' are syntactically independent of O' and the quantifier-free checks from Def. 10.3 are used. What is left to show is that no bad state is reachable in W. We will be considering the equivalence classes induced by O' on the state space of W, i.e., states only differing in O' are in the same equivalence class. Assume there exists a bad trace $s_0, \ldots, s_n$ in W. By the reset check, all states in the equivalence class of s_0 also satisfy R', and by the base check are guaranteed to be good. By applying the transition check to all states in the equivalence class of s_0, we know they each have a transition to all states in the equivalence class of s_1. This allows us to apply the step condition to all states in the equivalence class of s_1, concluding that they are all good. This argument can be applied n times to show that all states in the equivalence class of s_n are good – including s_n itself. □

This more complex certificate is, in fact, hard for the second level of the polynomial hierarchy.

Theorem 10.15. *A closed formula $\forall A \exists E f$ is true exactly if $W = (I', L', R', F', P', C')$ passes the Step$^{\exists}$ check, where $I' = A, O' = E, L' = \{\ell\}, r_\ell' = \top, f_\ell' = \top, C' = \top$, and $P' = \neg f \land \neg \ell$.*

Proof. In the Step$^{\exists}$ check, F' is simply ℓ_1, which implies $\neg P_1'$. With C constant true, O_0' quantifies over $\neg f_0 \land \neg \ell_0$. Since ℓ_0 is independent of O_0', we can leave it behind when moving the rest to the other side of the implication. Omitting the universal quantifiers for unused variables, we are left with: $\forall \ell_0 \forall \ell_1 \forall A_0 \, [\neg \ell_0 \land \ell_1 \rightarrow \exists E_0 \, [f_0]]$, which is trivially true for all valuations of the first two quantifiers, except for one where it simplifies to $\forall A_0 \exists E_0 f_0$. □

10.5.1 Uniqueness Constraints

In the following, we present certificate generation for uniqueness constraints. We begin by providing the definition of k-induction under uniqueness constraint.

Definition 10.16. A property P of a given circuit $M = (I, L, R, F, P, C)$ is said to be k-inductive under uniqueness constraints iff the following holds:

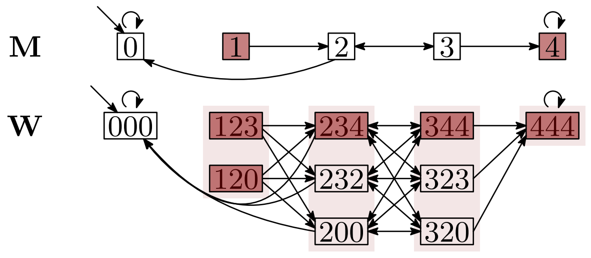

Figure 10.6: Example of the k-witness construction. The figure displays the state space of a circuit M, which is 3-inductive under uniqueness constraints, alongside its corresponding k-witness circuit W (Def. 10.17). Bad states are marked in red, state 0 is the only initial state. Witness states that differ only by the value of O' are grouped vertically. If at least one state in a group violates the property, the entire group is highlighted with a light red background.

$$R_0\{L\} \wedge \Big(\bigwedge_{i\in[0,k-1)} F_{i,i+1}\{L\}\Big) \rightarrow \Big(\bigwedge_{i\in[0,k)} \big(\bigwedge_{j\in[0,i]} C_j\big) \rightarrow P_i\Big) \text{ and}$$

$$\Big(\bigwedge_{i\in[0,k)} F_{i,i+1}\{L\}\Big) \wedge \Big(\bigwedge_{i\in[0,k)} C_i \wedge P_i\Big) \wedge \mathit{unique}_k \rightarrow (C_k \rightarrow P_k),$$

$$\text{where } \mathit{unique}_k = \bigwedge_{0\le i<j<k} (I_i \not\leftrightarrow I_j) \vee (L_i \not\leftrightarrow L_j).$$

The first formula, called the initiation check, specifies that the property holds for k steps from the initial states. The second formula is an inductive check such that if the property holds for k steps then it also holds at the $k+1$th step. The uniqueness constraint ensures that, in the inductive step, the k consecutive states are all distinct from one another.

To certify k-induction under uniqueness constraints, we can construct a witness circuit, using additional inputs as *oracles*. An example for an intuitive understanding of our construction can be found in Fig. 10.6. The construction relies on the following idea. A k-inductive property can be strengthened to be inductive: for every bad state, all predecessors up to k steps back are labelled as bad states too.

Definition 10.17 k-witness circuit under uniqueness constraints**.** Given a circuit $M = (I, L, R, F, P, C)$ with $k \in \mathbb{N}^+$. The k-witness circuit under uniqueness constraints is $W = (I \,\dot\cup\, O', L, R, F, P', C)$, where

- $O' = \bigcup_{i\in[0,k)} I^i$ and
- $P' = \bigwedge_{i\in[0,k)} [v^i \rightarrow P(I^i, L^i)]$, with
- $v^0 = C(I, L)$, $v^i = v^{i-1} \wedge C(I^i, L^i)$,
- $L^0 = L$ and $L^{i+1} = \{f_l(I^i, L^i) \mid l \in L)\}$.

In the above definition, $L^1, \dots, L^{k-1}$ are not necessarily latches: we directly use the output of the transition functions, which are simply AND-gates in practice. The variables v^i keep track of the depth up to which the path along the L^i remains valid; that is, only traverses states satisfying the constraint. The latches are the same in the original and the k-witness circuit, however, the k-witness circuit has more bad states, as it requires all states reachable in $k-1$ steps to be good. Note that by construction the reset and transition functions are independent of the oracles, thus the reset$^{\exists\forall}$ and transition$^{\exists\forall}$ checks are not necessary and the quantifier-free versions should be used.

Even though Def. 10.17 is also a valid construction for k-induction *without* uniqueness constraints, it is expensive for the certificate checker to perform QBF checks. Hence in the case where uniqueness constraints are not required, the witness circuit construction should still follow the construction outlined in [12]. Their approach produces inductive witnesses by recording a history of k states rather than precomputing a possible future. While this method avoids the quantification, it would violate the transition check, since the witness is not allowed to transition to a state already in the history while the correctness of the construction relies on the model being inductive under uniqueness constraints.

	A	O	Σ	$\cup$	Σ_P	$\cup_P$	Σ_F	$\cup_F$
Mean (39)	25411	64	777.20	37.18	36.23	30.77	660.37	4.15
picorv32 (8)	54042	201	3455.78	144.77	107.99	128.20	3037.94	14.89
mentor (1)	31613	36	505.96	38.70	23.40	25.53	440.87	12.37
dspfilters (17)	28397	49	120.07	9.45	29.66	7.58	58.22	1.22
sm98 (3)	5126	2	25.17	21.35	1.56	1.16	4.83	2.38
zipcpu (7)	2946	2	3.46	2.57	1.49	1.46	0.54	0.23
zipversa (3)	2775	3	5.92	3.44	2.17	2.13	0.70	0.26

Table 10.1: The HWMCC set contains 39 multi-property benchmarks, split into 7 families, each with the same number of properties (B). The other columns display the number of and-gates (A), total time taken in seconds for checking individual witness circuit for a single property (Σ), and time taken for checking the composed witness circuit ($\cup$). Additionally, the table presents the time taken up just by the Property check for the individual witnesses (Σ_P) and the composed witness ($\cup_P$), which in both cases has to be done B times. As well as the Transition check, which has to be done for each individual witness (Σ_F) but only once for the composed witness ($\cup_F$). This is the same for every other check, but the transition check is the most complex on this set of benchmarks. Each row presents the mean over the benchmarks in the family (number given in parentheses). The mean over all benchmarks is presented at the top.

Theorem 10.18. *Given a circuit $M = (I, L, R, F, P, C)$ with some $k \in \mathbb{N}^+$, and a k-witness circuit $W = (I \dot\cup O', L, R, F, P', C)$. If M is k-inductive under uniqueness constraints, then W is a valid witness circuit for M.*

	t_{MC}	t_{Cert}	t_{SAT}	k	I	A
bobcount	426.68	26.86	2.40	76	3	77
eijks298	383.05	2192.33	9.77	138	3	225
pdtvispeterson	6.29	249.35	2.76	59	2	700
pdtvisvending00	14.25	to	2.37	35	2	959
pdtvisvending02	199.05	to	2.03	33	2	958
pdtvisvending05	3.13	26663.62	1.70	28	2	951
pdtvisvending07	3.41	22117.90	2.34	29	2	952
pdtvisvending08	117.81	to	2.07	33	2	950

Table 10.2: In the HWMCC'10 benchmark set, 8 additional instances are solved with uniqueness constraints. We give the model checking (t_{MC}) and certification time (t_{Cert}). The table also presents the portion of the certification time taken for the 4 SAT checks (t_{SAT}). All times presented are in seconds. The last three columns denote the inductive depth k, the number of input variables I, and gates A respectively.

Proof. Since L, R, F and C remain unchanged, the reset and transition condition hold trivially. Given C, the new property P' clearly implies P, thus the property check is satisfied. For the base check we note, that if an initial state of the witness violates P', a bad state in the original circuit can be reached in k steps or less. It is left to show that if the step$^{\exists}$ check fails, the original circuit is not $k-$inductive. Assume two consecutive states u' and v' in C' for which the step$^{\exists}$ check fails. Let u and v be the states in M induced by the assignment to L in u' and v' respectively. The shortest path from v to a bad state has exactly k states, all are different, satisfy the constraint, and all but the last are good. If the number of steps was less, P' would not hold in u', and if it was any more, P' would hold in v'. The other two claims follow from the path being the shortest. Since u is guaranteed to satisfy C and be different from the bad state, prepending the path with u yields a path in M consisting of k unique good states followed by a bad state, thus M is not k-inductive under uniqueness constraints. □

10.6 Experimental Evaluation

We implemented the proposed certificate format in the open-source certificate checker CERTIFAIGER, together with the relaxed checks that require quantifiers. Note that in all HWMCC benchmarks, all reset functions are stratified, as the standard AIGER format used in the competitions only supports stratified resets. We also extended the k-induction-based model checker MCAIGER [270] to generate k-witness circuits as defined in Def. 10.17. Additionally, we implement our tool AIGMERGE[1] that uses the construction described in Theomerm 10.5 to compose arbitrary circuits. CERTIFAIGER utilizes KISSAT [30] for SAT checks and the QBF solverQUABS [292] for quantified checks. All input circuits are in the AIGER 1.9 format. Each certificate check is

[1]AIGMERGE may eventually be added to the set of AIGER utilities at https://github.com/arminbiere/aiger.

generated as combinatorial circuits in either AIGER (for SAT checks) or QAIGER (for QBF checks) and is then translated to CNF or the circuit-based QCIR format [293].

The goal of our first experiment is to evaluate the certification method for composing witness circuits, which also provides insight into certifying extracted constraints. Although we could not find an open-source model checker that implements explicit constraint extraction, the concept of model constraints naturally arises in circuits where multiple properties are of interest. Therefore, we focus on multi-property benchmarks that are present in HWMCC'12 and HWMCC'19. We ran the open-source certifying model checker VOIRAIG for each property individually, which left us with one witness circuit per property that could be proven. We then used AIGMERGE to combine all the witness circuits for an individual model into one. Certificate checking for such a certificate proceeds as follows: Verify the reset, transition, base, and step condition for the composed witness according to Def. 10.3. For each original property run the property check. We compare the total time of these checks, to total time it take to certifying the original witness circuits for each property individually. We present the results obtained in Table 10.1. As can be seen, directly verifying the composed witness circuit is significantly more efficient than verifying individual witness circuits for each property. Interestingly, the transition check appears to be the bottleneck during the certification process, differently from experimental results in [9] where the step check always take significantly longer.

In the second experiment, we study the effectiveness of our certification method for k-induction under uniqueness constraints. We selected the benchmarks that require uniqueness constraints from HWMCC'10. Results are summarized in Table 10.2. We used a timeout of 50,000 seconds for certificate checking. Despite the QBF check creating a bottleneck in certification performance, our certifier successfully validated 5 out of 8 instances. The QBF solver QUABS failed to complete the step$^{\exists}$ checks for three instances within the time limit, whereas for the same instances the rest of the checks were solved in less than 10 seconds (see t_{SAT} in the table). While QBF solving is inherently more challenging than SAT solving, we attribute the substantial performance gap in part to recent advances in SAT solving—particularly the circuit-specific optimizations introduced in KISSAT [1]. We believe that incorporating similar optimizations into QUABS could significantly improve the efficiency of QBF-based certificate checking. Exploring such enhancements is an important direction for future work.

10.7 Conclusion

Recent hardware model checking competitions have embraced a standardized certificate format, relying on simple SAT-based certificate validation. This approach has proven highly effective, with certification overhead constituting only a small fraction of the total verification effort. In this work, we introduce certification generation techniques tailored to both explicit and implicitly extracted constraints for three different constraint classes. Furthermore, we present alternative QBF-based checks to replace pure SAT checks in the certificate format, addressing the challenges of certificate generation posed by intricate reset logic and complex encodings in model checking. Extending this, we also introduce a certification approach for k-induction under uniqueness constraints. The empirical evaluation, conducted on a wide range of competition benchmarks, demonstrates the effectiveness and practical relevance of our method. Looking forward, we plan to incorporare the remaining techniques in bit-level hardware model checking, such as retiming [294] and localization. Furthermore, we aim to broaden the scope of our certification method to infinite-state systems to support program verification as well as exploring the certification of security properties.

Chapter 11

Introducing Certificates to the Hardware Model Checking Competition

Published In International Conference on Computer Aided Verification (CAV) 2025

Authors Nils Froleyks, Emily Yu, Mathias Preiner, Armin Biere, and Keijo Heljanko

Changes from Published Version Added additional experiments in Appendix 11.6. Corrected typos and made minor wording adjustments to improve flow.

Authors Contributions The author extended the certificate format to include constraints, proved its correctness, and implemented it in robust tooling for use in the competition. He co-organized the competition and supported participants in debugging their model checkers, allowing the competition to reach the success it did. He further produced the figures in the paper and conducted the supplementary evaluation.

Acknowledgement This work is supported in part by the ERC-2020-AdG 101020093, the LIT AI Lab funded by the State of Upper Austria, the Research Council of Finland under the project 336092, and a gift from Intel Corporation.

Furthermore we of course also owe a big thank-you to the submitters of model checkers and benchmarks to the competition over all these years. Without their enthusiasm and support neither the competition nor this study would exist.

Abstract Certification was made mandatory for the first time in the latest hardware model checking competition. In this case study, we investigate the trade-offs of requiring certificates for both passing and failing properties in the competition. Our evaluation shows that participating model checkers were able to produce compact, correct certificates that could be verified with minimal overhead. Furthermore, the certifying winner of the competition outperforms the previous non-certifying state-of-the-art model checker, demonstrating that certification can be adopted without compromising model checking efficiency.

11.1 Introduction

Competitions have played a key role in advancing the state of the art in automated reasoning tools by enabling direct performance comparisons across a wide range of solvers, offering challenging benchmarks, and fostering new research. However, many of these tools operate as black boxes by providing only *true* or *false* as an output. Certification addresses this limitation by requiring a counterexample when verification fails and a proof when it succeeds. Since certificates can be independently validated, they significantly enhance confidence in the correctness of verification results, thereby improving the reliability of solvers.

One goal of using certificates in hardware model checking is to repeat the success story of proof certificates in SAT for this automated reasoning domain with a large industrial user base. Besides increasing trust in verification results, certificates enable more complex design optimizations, allow to continue using legacy code and can streamline and improve efficiency of tool development in both verification and synthesis. The simple proof certificate format used in SAT still allows to capture a wide range of solving optimizations at industrial scale. In this case study, we investigate whether the simple model checking certificate format employed in the recent hardware model checking competition has the potential to achieve the same for hardware model checking.

The Hardware Model Checking Competition (HWMCC) has its roots in a rather lively discussion at the 2nd Alpine Verification Meeting (AVM) in 2006 among Daniel Kröning, Dirk Beyer and Armin Biere. The debated question was how model checking research as well as industrial applications can benefit from competitions in the same way the SAT competitions were instrumental in advancing SAT. Daniel Kröning and Armin Biere argued to focus on hardware gate-level models with simple and clear semantics.[1]

This argument prompted the development of the AIGER format [50] used in the first (hardware) model checking competition, affiliated to CAV'07. This first version of AIGER (20071012) came with a library for parsing and other essential tools, including a translator from SMV and BLIF to AIGER. The challenge of the first competitions in 2007, 2008 and 2010 was to collect benchmarks.

For the 2011 competition the first major revision of the AIGER 1.9 format [51] included *liveness* properties and *constraints*. The following competitions from 2012–2015 [295] and in 2017 [160] included a *deep bound track* to emphasize the common industrial practice of relying on incomplete but deep bounded model checking. In 2019 a word-level track was established based on the BTOR 2.0 format [296] proposed at CAV'18. After focusing on word-level in 2020 the organizers decided in 2024 [21] to reintroduce a bit-level track but take the chance to force all participating model checkers to produce certificates.

The introduction of mandatory certification in HWMCC'24 significantly impacted participation and competition dynamics. The 2024 competition saw a record nine participants, up from three in the previous edition, reflecting growing interest and

[1]Dirk Beyer proposed to use C as input language, which is much harder to master, due to its complex semantics. Accordingly the first SV-COMP took place in 2012.

accessibility. Discussions with participants revealed that new rules, *particularly the requirement for certification*, leveled the playing field by encouraging the development of verifiable solvers. Feedback indicated that participants successfully implemented certificate generation based on our published results [9, 12, 13, 253]. It was also noted that implementing correct model checking algorithms demanded substantially more effort than generating certificates.

The certificate format itself has undergone several iterations with the ultimate aim of its use in the competition. In HWMCC'24, all participating model checkers were required to produce proofs alongside the model checking results for both safe and unsafe instances. For unsafe instances, the certificate is a trace serving as a counterexample, which can be validated via simulation; as for safe instances, it is a proof witness circuit. For the competition we use an extended version of the witness format defined in [9], that supports constraints, an essential feature of AIGER 1.9.

We first describe the certificate format used in the competition, then present experimental findings. We investigate the overhead introduced by certificate checking in model checking and results show that it accounts for only *a fraction of the total verification time*. Moreover, we compare the certifying winner of HWMCC'24, RIC3, against the state-of-the-art model checker ABC which does not support certificate generation. And see that even when including the time for witness validation, RIC3 outperforms ABC.

11.2 Related Work

Certification in other competitions. Certification has been an essential part in many other competitions. In SAT competitions [14], certification has been mandatory for almost a decade, as a fundamental requirement. Solvers must produce certificates for both SAT and UNSAT instances: a satisfying truth assignment for SAT and a proof in the DRAT [101] format for UNSAT. A solver is disqualified from the main track if a single certificate is found invalid. The software verification community is following suit. At SV-COMP'24, it is the second year of having a dedicated track for witness validation, with a range of participating witness validators [297]. The MaxSAT Evaluation [298] has also taken a step forward in 2024 by requesting proofs for the first time. In QBF Evaluations [299], there used to be a dedicated Evaluate & Certify track, where solvers are required to produce proofs that are easier to check than the solving task; however, as the organizers pointed out, only a few QBF solvers support certificate generation. SMT competitions (SMT-COMP) [300] and ATP System Competitions (CASC) [301] feature a wide variety of theories and have yet to adopt a universal certification standard. Classical Planning is similar to verification, but usually more focused on finding solutions (plans). Nevertheless, a deductive certificate format [302] has been introduced, and extended to support UNSAT certificates produced by an underlying SAT solver [303].

Related work in model checking certification. Deductive proof systems have been used for generating proofs of model checking. For example, the author of [246] addresses μ-calculus, while the authors of [243] focus on liveness and several pre-processing techniques. These approaches require model checkers to provide deductive proofs. The works in [289, 290] explore the use of inductive invariants as certificates for k-induction. Notably, the certificate format employed in HWMCC'24 is also compatible with these inductive invariants. The authors of [244] use liveness-to-safety reduction techniques to certify liveness properties. The problem of certifying model checking has also been addressed in infinite-state systems [245, 304] where SMT solvers are leveraged for unbounded state spaces. An alternative approach to providing certificates, is to formally verify the model checker itself, as demonstrate in [237].

11.3 Certificate Format

We assume the standard notions and terminology of Boolean logic. In the following, we consider hardware designs modeled as Boolean circuits encoded as sequential and-inverter graphs (AIGs)[50, 51, 57, 262]. Such a Boolean circuit is given as a tuple $M = (I, L, R, F, P, C)$ where I is an ordered set of inputs, L is an ordered set of latches, R defines the set of reset states, represented as a reset predicate that holds when every latch $l \in L$ equals its reset function r_l; F is the transition predicate that refers to two consecutive states, and encodes that each latch in one state is equal to its corresponding transition function f_l applied to the previous state; P and C are predicates that define the set of good states and the set of states valid under the *constraint*, respectively. These predicates, along with the reset and transition functions, are encoded in the circuits as binary AND-gates with possible negation at the incoming gates.

We use the notion of a reset predicate R being stratified; for space reasons we refer to [12] for formal definitions. In essence, it means that the dependencies that the reset functions introduce among the latches are acyclic. For $K \subseteq L$, $R\{K\}$ and $F\{K\}$ restrict these predicates so only the latches in K are required to be equal to their reset or transition. When referencing a sequence of states, we use indices on the predicates to represent the corresponding copy of the predicate at a certain state in the sequence.

A trace of length n is a sequence of $n+1$ states, where the first state needs to satisfy R, every pair of consecutive states satisfies F (written $F_{i,i+1}$ for the i-th and its successor state) and all states satisfy the constraint C. If the last state violates P, the trace is bad. Thus a satisfying assignment to the following formula *certifies* that a circuit is *unsafe*:

$$R_0 \;\wedge \bigwedge_{i\in[0,n)} F_{i,i+1} \;\wedge \bigwedge_{i\in[0,n]} C_i \;\wedge\; \neg P_n.$$

For *safe* instances, the certificate format employed in HWMCC'24 takes the form of witness circuits, defined as follows.

Definition 11.1 Witness Circuit**.** The circuit $W = (I', L', R', F', P', C')$ is a witness circuit of $M = (I, L, R, F, P, C)$, if R' is stratified and for $K = L \cap L'$:

1. Reset: $R\{K\} \wedge C \;\Rightarrow\; R'\{K\} \wedge C'$;
2. Transition: $F_{0,1}\{K\} \wedge C_0 \wedge C_1 \wedge C'_0 \;\Rightarrow\; F'_{0,1}\{K\} \wedge C'_1$;
3. Property: $(C \wedge C') \;\Rightarrow\; (P' \;\Rightarrow\; P)$;
4. Base: $R'\{L'\} \wedge C' \;\Rightarrow\; P'$;
5. Step: $P'_0 \wedge F'_{0,1}\{L'\} \wedge C'_0 \wedge C'_1 \;\Rightarrow\; P'_1$.

The five conditions described above are simple SAT checks. An additional polynomial-time check is required to verify that R' is stratified. If all checks pass, M' is a valid certificate for M, certifying its safety property. The first three conditions in Def. 11.1 establish a simulation relation between two circuits, such that if M' is safe, M is also safe. Intuitively, an initial state in the original circuit M corresponds to an initial state in the witness circuit, and each valid transition in M corresponds to a transition in M'. Property P' is a strengthening of P. Consequently, safety of M' implies safety of M. In summary, a bad trace in M corresponds to a bad trace in M'. A sketch of the traces for both M and M' is provided in Fig. 11.1. The latter two checks (Def. 11.1.4 and Def. 11.1.5) prove P' to be an inductive invariant, entailing the safety of M'. We provide a high-level intuitive illustration of Def. 11.1 in Fig. 11.1.

This is a slight extension to the format in [9], as it supports constraints and now covers all AIGER 1.9 [51] features except liveness. In HWMCC'24, witness circuits are also produced as AIGER files. The witness circuit validation is implemented in the certificate

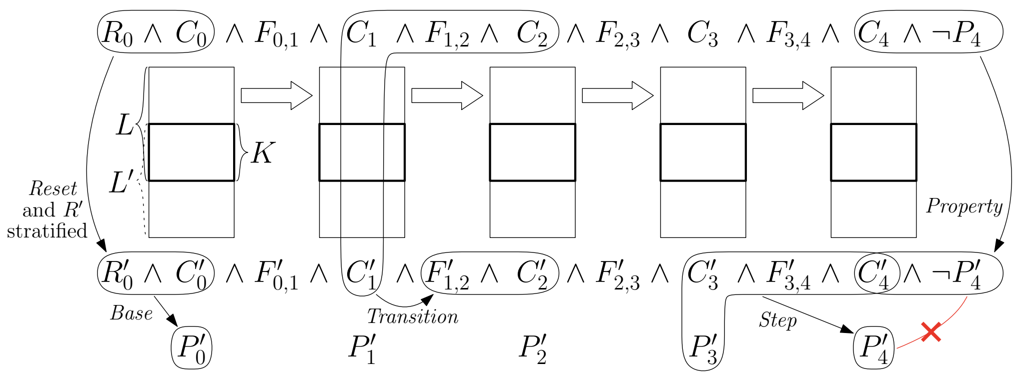
$R_0 \land C_0 \land F_{0,1} \land C_1 \land F_{1,2} \land C_2 \land F_{2,3} \land C_3 \land F_{3,4} \land C_4 \land \neg P_4$
L
K
L'
Reset and R' stratified
Property
$R'_0 \land C'_0 \land F'_{0,1} \land C'_1 \land F'_{1,2} \land C'_2 \land F'_{2,3} \land C'_3 \land F'_{3,4} \land C'_4 \land \neg P'_4$
Base
Transition
Step
P'_0
P'_1
P'_2
P'_3
P'_4

Figure 11.1: An illustration for the correctness of Def. 11.1. Assuming that a circuit M with a valid witness M' has a bad trace leads to a contradiction. Depicted are the overlapping sets of variables and how conditions of the witness check are used to construct a bad trace in M', and arrive at a contradiction. For *Transition* and *Step* only one application is illustrated.

checker CERTIFAIGER[2] used for the competition, but has not been described in detail before. For efficient certification, CERTIFAIGER leverages the SAT solver Kissat 4.0.0, winner of the SAT competition 2024.

11.3.1 Soundness of the Certificate Format

We present a proof that the existence of a witness circuit as defined in Def. 11.1 indeed certifies the safety of a model. The proof extends what is presented in [9] by considering constraints.

Theorem 11.2. *Given two circuits M and M', with $M = (I, L, R, F, P, C)$, and $M' = (I', L', R', F', P', C')$. If M' is a valid witness circuit for M, then M is safe.*

Before proving the main theorem, we first introduce some additional notation: An assignment maps a subset of the gates to true or false, and is always consistent with the valuation of the AND-gates. Extending an assignment means assigning more gates while leaving previously assigned gates unchanged. We refer to the reset gate associated with latch l as r_l and the primed version r'_l, when referencing the reset gates used by R'.

Every gate g refers to the Boolean function defined by its fan-in cone, and we write $g(s)$ to denote that we consider the function under an assignment s, i.e., the variables in g which are assigned by s are replaced with the corresponding constants. A function g *semantically depends* on a variable v if an assignment exists under which $g(s_v)$ and $g(s_{\neg v})$ evaluate to different truth values.

We first show that a reset state in M corresponds to a reset state in M'.

[2]https://github.com/Froleyks/certifaiger

Lemma 11.3. *For circuits $M = (I, L, R, F, P, C)$ and $M' = (I', L', R', F', P', C')$ satisfying the reset check (Def. 11.1.1) and R' stratified, any assignment to $I \cup L$ satisfying $R\{K\} \wedge C$, where $K = L \cap L'$, can be extended to satisfy $R'\{L'\} \wedge C'$.*

Proof. Assuming the reset check passes and R' is stratified, let s be an arbitrary but fixed assignment to $I \cup L$ satisfying $R\{K\} \wedge C$. The assumptions of the Lemma further imply that s satisfies $R'\{K\} \wedge C'$. To show that s can be extended to satisfy $R'\{L'\}$, we first prove for each latch $l \in K$, $r'_l(s)$ has no semantic dependency outside $(I \cup L) \cap (I' \cup L')$. Assume, for contradiction, there is a latch $l \in K$ with $r'_l(s) \not\Leftrightarrow r'_l(s_u)$ where s_u is the same as s except for the value of some gate $u \in (I' \cup L') \backslash (I \cup L)$. We have $l \not\Leftrightarrow r'_l(s_u)$ and therefore $R'\{K\}$ does not hold under s_u. However, u is not in $I \cup L$ and $R\{K\} \wedge C$ still evaluates to true under s_u, thus implying $R'\{K\}$, and leading to the desired contradiction.

Since R' is stratified, the semantic dependencies of the reset gates r'_l can be seen as a topologically sorted graph. Given the above result, when considering $r'_l(s)$, the remaining dependency graph can be sorted topologically such that the variables in $(I \cup L) \cap (I' \cup L')$ are at the bottom. Thus, s can be extended to satisfy $R'\{L'\}$ by assigning the remaining latches in the reverse of that order. The extended assignment still satisfies $R\{K\} \wedge C$ and thereby C'. □

We can now move on to prove the correctness of the certificate format, i.e., the proof of the main Theorem 11.2. Refer to Fig. 11.1 for a visualization of the proof.

Proof. Suppose, for contradiction, M is unsafe. Then there is a bad trace of some finite length n in the form of an assignment to $n + 1$ copies of $I \cup L$ satisfying:

$$R_0\{L\} \wedge C_0 \wedge F_{0,1}\{L\} \wedge C_1 \wedge \cdots \wedge C_{n-1} \wedge F_{n-1,n}\{L\} \wedge C_n \wedge \neg P_n.$$

We extend this assignment to each copy of the gates in $I' \backslash I \cup L' \backslash L$ that satisfies:

$$R'_0\{L'\} \wedge C'_0 \wedge F'_{0,1}\{L'\} \wedge C'_1 \wedge \cdots \wedge C'_{n-1} \wedge F'_{n-1,n}\{L'\} \wedge C'_n \wedge \neg P'_n.$$

Let $X' = (I' \cup L') \setminus (I \cup L)$. The assignment satisfying $R_0\{K\} \wedge C_0$ can by Lemma 11.3 can be extended to X'_0 satisfying $R'_0\{L'\} \wedge C'$. With that and the transition check $F'_{0,1}\{K\} \wedge C'_1$ is satisfied and the assignment can be extended to X'_1 satisfying $F'_{0,1}\{L'\} \wedge C'_1$ by the definition of transition functions.

Applying the same argument n times yields an assignment to $(I \cup L \cup I' \cup L')^n$ satisfying $F'_{i,i+1}\{L\}$ for $i \in [0, n)$ and C_i for $i \in [0, n]$. Lastly, the property check guarantees $\neg P'_n$, giving us the desired assignment. However, the base and step check together ensure that the property P' holds on all reachable states of M', thus contradicting the initial assumption that a bad trace exists in M. □

11.4 Evaluation

In this section, we present a comprehensive analysis of the competition results [3], focusing on the overhead of certificate generation and checking. Specifically, we address the following three questions:

1. What is the runtime overhead associated with validating certificates?
2. What is the space overhead associated with storing certificates?
3. How do certifying model checkers compare to the state of the art?

Experimental Setup. The 2024 competition ran on a cluster of 48 compute nodes equipped with an AMD Ryzen 9 7950X 16-core processor at 4.5 GHz and 128 GB or RAM, running Ubuntu 20.04 LTS. For fairness, the experiment described in Section 11.4.3 ran on the cluster used for the last competition in 2020. Each node has access to two Xeon E5-2620 v4 CPUs, for a total of 16 cores running at 2.1 GHz, and 128 GB of RAM.

We focus on (all) the 319 bit-level benchmarks of HWMCC'24, which were translated from the word-level (BTOR/bit-vector) track of HWMCC'24. The majority of the benchmarks (250) are new benchmarks submitted in 2024 by three different groups, including benchmarks for checking safety properties of open source RISC-V cores, sequential equivalence checking, branch coverage problems, as well as software verification problems, which were translated from SV-COMP'24 [297]. The remaining 69 benchmarks were selected randomly from previous competition years (2019 and 2020). Each model checker had exclusive access to a node, with a 120 GB memory limit and a one-hour wall-clock limit. A separate limit of 10 hours was imposed for certificate checking.

Note that for precision and reliability of measurements, the competition cluster uses `runexec` to measure resource consumption of the model checkers. We further rely on it to properly isolate the processes and to enforce both the time and memory resource limits.

11.4.1 Certificate Checking Overhead

We now evaluate the overhead introduced by certificate checking. For each solver, we consider the model checking time, t_{MC}, the time required to validate the produced certificate, t_{CERT}, and the total time $t_{\text{TOTAL}} = t_{\text{MC}} + t_{\text{CERT}}$. The certificate checking overhead for a model checker refers to the additional time required to run all benchmarks when certification is enabled. Note that benchmarks unsolved by the model checker are excluded from this metric. The results are displayed in Figure 11.2 where both safe and unsafe instances are considered.

The clear winner of the competition is RIC3, demonstrating superior performance on both safe and unsafe benchmarks. When considering only safe benchmarks, the

[3] https://hwmcc.github.io/2024

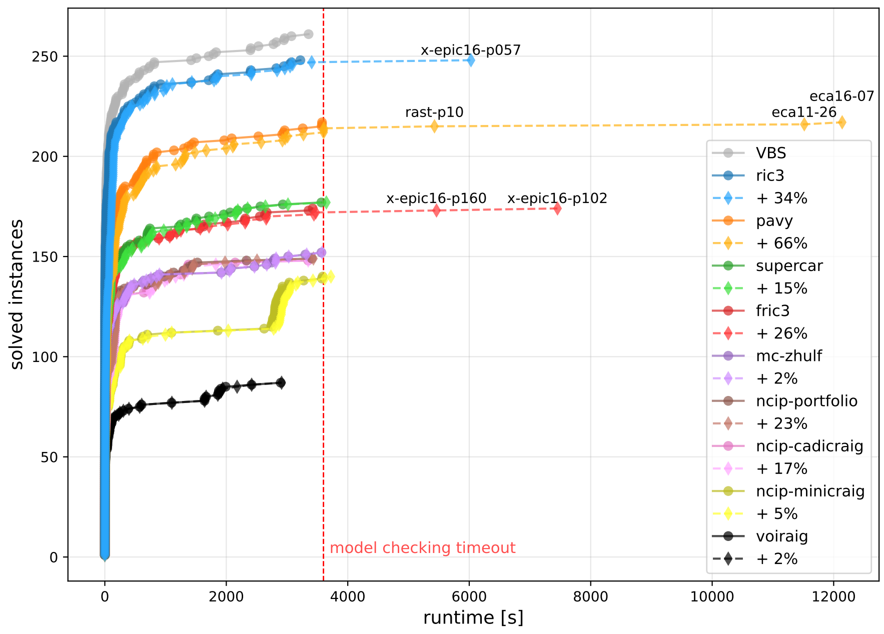

Figure 11.2: HWMCC'24 results (319 benchmarks). The plots show the number of solved instances as a function of time. For each model checker, we present (1) the model checking time and (2) the total time for model checking and certificate validation. Diamonds represents the time taken to model check a circuit and validate the produced witness, while dots indicate model checking time only. Benchmarks whose certificates were especially time-consuming to verify are labeled. The Virtual Best Solver (VBS) indicates the top solver performance on each instance. The legend includes the overall certification overhead. The results clearly indicate, that certificate validation only adds minimal overhead.

ranking remains virtually unchanged, with FRIC3 narrowly outperforming SUPERCAR. As for unsafe instances, which constitute approximately 30% of solved benchmarks, SUPERCAR slightly outperforms PAVY. In both scenarios, RIC3 maintains its lead and performs impressively close to the virtual best solver.

In Figure 11.2, we also identify six outliers where the combined model checking and certification time exceeded the one-hour model checking timeout by more than 5%. The difficulty in their certification seems to be related to the witness circuit generation process *within* the model checker, as for each of these instances, another model checker found a witness circuit, which could be validated under 100 seconds. An exception is the x-epic16-p057 benchmark, which was solved exclusively by RIC3. Certificate checking never exceeded the 10-hour limit.

As Figure 11.2 shows, the overall certification overhead only gives rise to a small fraction of the model checking time, which is highly promising and highlights the effectiveness of the certificate format. For instance, when using RIC3 to model check

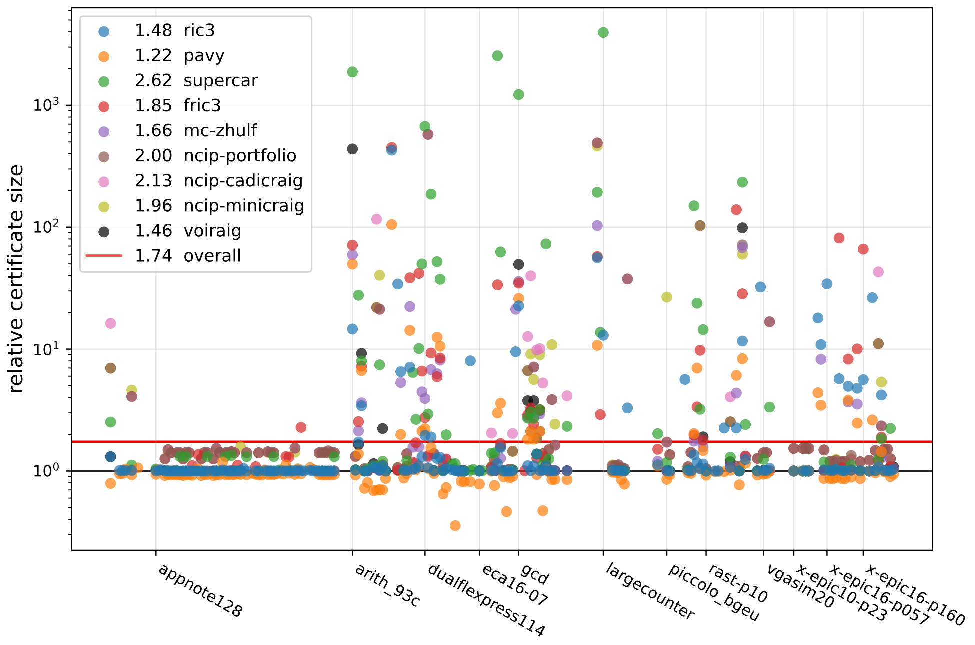

Figure 11.3: Size of produced witness circuits relative to their original model circuit. The x-axis represents the set of benchmarks, sorted alphabetically, whereas the y-axis indicates the certificate size (gates) relative to the model. The legend also shows the geometric mean of the relative certificate size for each model checker and all produced certificates combined. Dots stacked vertically correspond to the same benchmark. Since the x-axis is sorted by benchmark name, neighboring instances are likely to belong to the same family. For clarity and space reasons, only a select few benchmarks are explicitly labeled. We observe an overall relative certificate size of 1.74, which indicates the compactness of the certificates.

all 248 instances it solved, the total time is increased by only 34% when all produced certificates are validated. In general, validating certificates for safe instances is a more challenging task than validating simulation traces for unsafe ones, a trend similar as in SAT solving. In fact, simulation time accounts for only 2% of the overhead in RIC3, and even less for all other solvers.

11.4.2 Certificate Size

Next, we evaluate the size of witness circuits for safe instances, where circuit size is measured in terms of gates, which includes the number of inputs, latches, and AND-gates. The relative certificate size is defined as $\frac{\text{certificate size}}{\text{model size}}$. Figure 11.3 presents the relative certificate sizes for all solved instances. Note that appnote and x-epic families, comprising 52 and 13 benchmarks respectively, depicted in the plot, include several multi-property benchmarks. In these cases, the benchmarks represent the same model, differing only in the property to be checked.

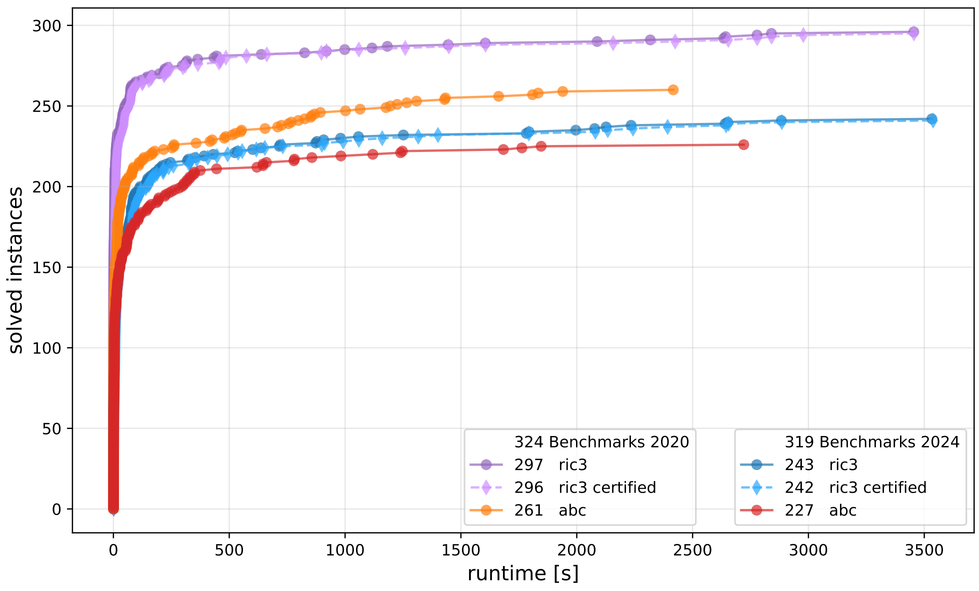

Figure 11.4: Comparing RIC3 (2024 winner) with ABC (2020 winner). The same HWMCC'20 hardware setup is used, for both benchmarks sets (2020 and 2024). RIC3 is also run in a fully certified mode, where each result is confirmed by checking the certificate. Note that for these certified runs the shown run-time not only includes model checking time but also certificate production and certificate checking time. We observe that on both sets RIC3 consistently outperforms ABC, even when accounting for certificate validation time.

We observe that PAVY produces smallest witnesses, with a geometric mean ratio of 1.22, whereas SUPERCAR exhibits the highest ratio of 2.62. Overall, more than 80% of the produced witnesses are less than twice as large as the certified model, with a geometric mean ratio of 1.74 across all produced witness circuits.

It further turns out that PAVY consistently generates witnesses substantially smaller than their corresponding models. Notably, this was not possible in earlier versions of the certificate format [12, 13, 253], which required the entire model to be embedded within the witness circuit. The original format was revised in [9] and went through another update for the competition, which is described in Sect. 11.3. This version allows, beside constraints, optimized witnesses that focus on a subset of the certified model, enabling significant reduction in witness size.

Witness size only correlates weakly with validation time. The biggest witness, produced by SUPERCAR for the largecounter benchmark, contains over 7 million gates for a model with fewer than 2 thousand gates, yet is verified within 900 seconds, which is 30% faster than model checking. Conversely, the two difficult-to-check eca witnesses produced by PAVY are 20% smaller than the model.

11.4.3 Comparison to State of the Art

Since participating model checkers in HWMCC'24 generate certificates, i.e., are *certifying model checkers*, it remains to show data on how certificate generation affects solver speed. We thus compare RIC3, the HWMCC'24 winner, with the state-of-the-art model checker ABC, the winner of the previous HWMCC edition in 2020, where witness circuits were not yet introduced. It is worth noting that the industrial-strength model checker ABC has dominated the bit-level track of the HWMCC since its debut in 2008. However, it could not participate in the 2024 competition, as certificates are now mandatory.

We use the version of ABC, which was submitted to the HWMCC'20, and was tailored specifically for the competition thus distinct from its public releases. To ensure that ABC is used with the same hardware specifications as expected by the participants in 2020 we run our experiment on the HWMCC'20 hardware. Note that this hardware is significantly older than the cluster used for HWMCC'24.

Note that the benchmarks from HWMCC'20 and HWMCC'24 were both included (there was no competition in between). The two sets are mostly distinct with only 8 benchmarks in common. This is following the SAT competition practice: HWMCC uses mostly new benchmarks every year, adhering to the SAT Practitioner's Manifesto.

Figure 11.4 shows that RIC3 convincingly outperforms ABC on both benchmark sets. Notably, in 2020, RIC3 solves 36 more benchmarks and is faster on 247 out of the 256 benchmarks solved by both model checkers. For RIC3, we also include a certified version, which represents its performance if it did internal certificate validation, and every benchmark is only reported as solved after the certificate has been successfully validated. Even in its certified mode, RIC3 still holds a clear lead, losing only one instance per year due to certificate validation exceeding the remaining time before the one-hour model checking timeout.

One minor exception is the performance on the 2024 benchmarks within the first 30 seconds, where the certificate checking adds a significant enough overhead for ABC to catch up to the certified version. Nevertheless, certificate production introduces *no measurable overhead* to overall model checking performance. These results demonstrate that RIC3, is a robust and efficient model checker, that presents superior performance while providing added benefits of certifying.

Invalid certificates. In HWMCC'24 and the experiments presented above, producing an invalid certificate causes the benchmark to count as unsolved. Out of the 1536 certificates generated during the competition, 44 were found to be incorrect. They were produced by four model checkers: SUPERCAR (20), NCIP-MINICRAIG (9), NCIP-PORTFOLIO (8), FRIC3 (7). The incorrect certificates produced by SUPERCAR are all simulation traces, notably 8 of them are for benchmarks which have been proven *safe* by other model checkers. In addition to 3 more incorrect simulation traces from FRIC3, all other invalid certificates were witness circuits failing one of the checks outlined in Def. 11.1.

Many of the invalid certificates stemmed from bugs uncovered by the organizers before the competition through extensive fuzz testing. The fuzzer and subsequent delta-debugging helped identify minimal failing circuits, shared subsequently with the model checker developers for fixes. Initially, all model checkers produced invalid certificates. After extensive feedback, most solvers passed thousands of fuzzer-generated test cases with correct certificates. This process highlights the benefits of certifying model checkers to improve their robustness.

Summary of results. Our experimental evaluation entails the following key findings. (i) Minimal overhead: certification adds only a small runtime overhead, representing a fraction of the total model checking time. (ii) Compact certificates: optimized certificate formats reduced storage requirements, with over 80% of certificates being less than twice the size of the certified model. (iii) Impact on performance: RIC3, the 2024 winner, outperformed the 2020 winner ABC, even when all certificates are verified, demonstrating that certifying approaches can simultaneously provide correctness guarantees and strong performance.

11.5 Conclusion

HWMCC'24 marks the first time that the Hardware Model Checking Competition has mandated certification for all participating solvers. Our case study confirms that certification can be integrated with minimal overhead while significantly improving confidence in verification results, illustrating the practical benefits of mandatory certification in hardware model checking.

Looking ahead, we call on more participants and model checker developers—both in academia and industry—to adopt and support certification. Building on the success of HWMCC'24, we intend to extend certification to the word-level track, for which a certificate checker CERBOTOR is already publicly available. However, challenges remain, including the need to develop techniques for generating certificates tailored to word-level-specific methods and addressing the use of trustworthy SMT solvers, which require SMT-based certificates.

On the other hand, a certifying liveness track is under planning, although this endeavor requires certificate generation for liveness checking algorithms, which remains another open research challenge. Another direction concerns the degree of trust we can place in the certificate checker. Ultimately, achieving a fully verified certificate checker would ensure an end-to-end correctness in the verification process, further increasing confidence.

Beyond increasing trust in model checkers, certificates have broader applications. An ongoing industry collaboration explores the integration of certifying model checkers as hammers in interactive theorem provers such as Isabelle [273] via Sledgehammer [274]. This entails the theorem prover encoding an open proof as a model checking problem, invoking a model checker, and lifting the certificate back into the theorem prover.

11.6 Appendix

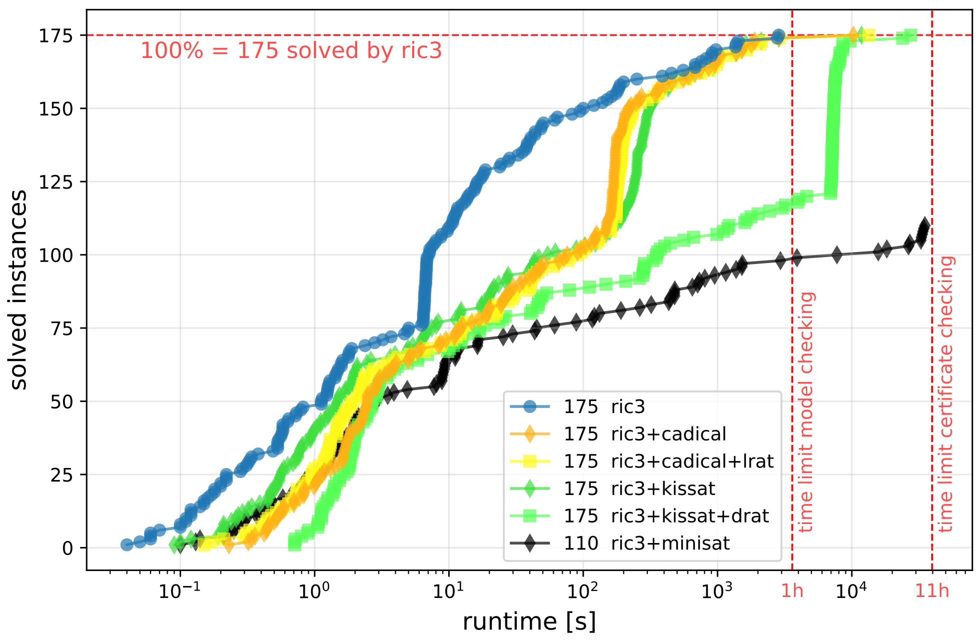

Figure 11.5: Comparison of Kissat and CaDiCaL, and the additional overhead incurred by checking DRAT and LRAT proofs respectively. MiniSAT is included as a baseline.

11.6.1 Base of Trust

So far, we have not discussed the *base of trust* of our certification approach. Besides the correctness of Theorem 11.2 and the implementation of Def. 11.1 in our certificate checker CERTIFAIGER, we also trust the SAT solver, in our case Kissat [30]. We can remove the SAT solver from the base of trust by additionally checking the DRAT [42] proof of unsatisfiability generated by the solver. In that case, we only trust the checker verifying the DRAT proof. However, even when using an unverified and efficient implementation such as DRAT-trim, proof checking can incur a significant overhead. Therefore, the LRAT proof format [43] which includes annotations to make proof checking more efficient has been developed. Currently the most efficient SAT solver which can produce LRAT proofs is CaDiCaL [4]. Figure 11.5 compares the two SAT solvers and the additional overhead incurred when also checking the proofs of unsatisfiability using DRAT-trim and LRAT-trim respectively.

11.6.2 SAT Solving Techniques

The effectiveness of our certification approach heavily relies on the efficiency of the underlying SAT solver. Recent advances in SAT solving techniques have made tackling formulas, encoding circuit structures in general, and checks as defined by Def. 11.1 specifically, significantly more efficient.

We focus on two techniques: *Clausal Congruence Closure* [1] and *Clausal Equivalence Sweeping* [1]. Clausal congruence closure extracts gate-level information from a

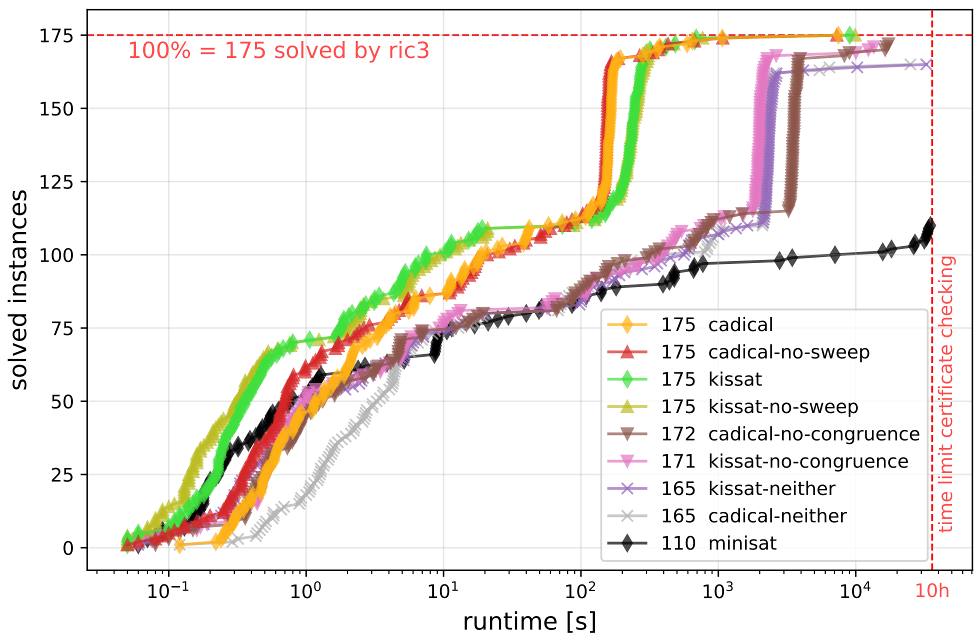

Figure 11.6: Comparison of the performance of Kissat and CaDiCaL with and without the Clausal Congruence Closure and Clausal Equivalence Sweeping techniques.

given CNF formula, thus reconstructing the original circuit structure to a certain extent. It then computes the congruence closure of the gate structure. This alone allows the SAT solver to instantly solve equivalence checking problems for isomorphic circuits, which is particularly relevant as the Reset and Transition checks in Def. 11.1 are very similar in structure to such equivalence checking problems if the witness circuit encodes just an inductive invariant, without adding additional sequential behavior.

Clausal Equivalence Sweeping implements the well-known circuit-level reasoning technique of SAT Sweeping at the CNF level. The original SAT sweeping algorithm recursively establishes equivalences between gates, by initializing a set of candidates using random simulation, and then calling an external SAT solver to prove equivalences. In Clausal Equivalence Sweeping, these steps are performed entirely at the CNF level, by extracting locally connected SAT formulas and using a simpler internal SAT solver to solve them.

So far Kissat and CaDiCaL are the only solvers implementing these techniques. The importance of these techniques to the task of checking witness circuits is illustrated in Figure 11.6.

Chapter 12

Conclusion and Future Work

We posed three research questions in the introduction. Each of them can be answered conclusively by the resounding success of the Hardware Model Checking Competition 2024 alone:

RQ1 Without the circuit-specific optimizations in our SAT solver, certificate validation would not have been nearly as efficient.

RQ2 The competition saw the highest number of participants to date, all of them fully certified. The top-performing model checker outperforms the previous state of the art, even when including certificate validation time.

RQ3 The introduction of our certification approach to the competition conclusively validated its practicality. It established a standardized format adopted by many model checkers, laying the foundation for fair and correct comparison of model checking techniques and their implementations.

Nine model checkers participated in the 2024 competition. Since most are composed of multiple engines, the number of distinct model checking algorithms was even higher. All of them produced fully certified results. This highlights the utility and applicability of our certification framework from the standpoint of model checking developers. The outstanding performance of the top solvers underscores the low overhead of certificate generation. Even more significantly, the remarkably low cost of certificate validation demonstrates the practical viability of our approach. These results clearly demonstrate, both to the academic community and to industry, that certification in model checking is not only possible—but practical. In doing so, we conclusively answered Research Question 2: how to increase trust in model checking.

This success would not have been possible without other foundational work on SAT solving. Validating model checking certificates itself is a highly specific application of SAT solving. Not only do the formulas all share a circuit structure, but the three checks establishing simulation are structurally very similar to combinatorial miters. The implementation of clausal congruence closure and SAT sweeping as integral components of the SAT solver has been crucial to the efficiency of certificate checking. These circuit-specific techniques improve not only certificate checking but also the performance of any model checker using a SAT solver that implements them. Furthermore, the development of LRAT proof logging and its mature implementation in CaDiCaL not only improves trust in the certification process but can also be used to reduce certificate validation time.

More relevant to model checking itself, we identified key challenges that the prominent IC3 algorithm poses to incremental SAT solving. Nearly every model checker in the competition employed at least one IC3 engine, and several of them use CaDiCaL. We extended the incremental SAT interface to better support large numbers of temporary clauses, enabling more efficient solving and simplifying solver usage. This extension has since been integrated into our backbone extraction tool, CadiBack, where it plays a critical role in ensuring performance. It has also been adopted by the wider community and may become part of the next version of the standardized interface for incremental SAT solving. These are among the aberrant applications we identified and improved SAT solving for while addressing Research Question 1.

CadiBack was initially developed to meet the needs of one of the authors. Since then, it has been widely adopted by the model counting community and is now a standard preprocessing component in many solvers. We further used it to enhance ternary simulation, a foundational technique in model checking, which we generalized into *cube simulation*. We formalized cube simulation as an instance of abstract interpretation and used it to generalize sophisticated preprocessing methods such as phase abstraction and temporal decomposition. The certificate constructions we proposed and proved complete incidentally verify not only the original methods but also our extended techniques.

The diverse set of certificate constructions detailed across multiple publications [9, 10, 12, 13, 253] —together with their open-source implementations in our model checker— serve as a comprehensive reference for anyone developing a hardware model checker, be it with or without certification.

Finally, we emphasized the essential role competitions play in scientific progress, thus answering Research Question 3. Hosting such competitions requires significant resources: both the computational capacity to run all participating tools on an even playing field and the human effort to set up, execute, and evaluate results. Nevertheless, competitions are crucial for any field which features any degree of practical applicability. They provide an objective proving ground for proposed ideas, beyond what any single group of researchers can achieve.

The SAT competition has long advanced not only solver efficiency but also the widespread adoption of certification. By introducing certificates to the hardware model checking competition, we took a decisive step in bringing it to the same standard. Not only will the trust in the results of the competition be significantly improved going forward, but by standardizing a unified format for all hardware model checkers, their use has been made simpler and safer.

During my doctoral studies, I not only contributed to theoretical foundations but also invested significant effort into developing robust tools for the community. Foremost among these is the certificate checker used in the Hardware Model Checking Competition 2024. We also published tools that implement the same certificate validation checks for two additional formats: Btor2 for word-level model checking, and dimspec for hardware designs not easily expressed in the AIGER format. In addition, my contributions to CadiBack and CaDiCaL will continue to serve the community for years to come.

Future Work

Several promising directions for future research emerge from this work:

Incremental Backbone Extraction. Currently, our backbone extraction tool [6] is separate from our SAT solver [4]. However, both share a largely similar API. Integrating backbone extraction into CaDiCaL would simplify its use across a wider variety of applications. Moreover, this integration would add support for incremental backbone extraction. Specifically, backbones under assumptions may pose interesting challenges and open up new applications.

Spatial IC3. We plan to explore new algorithmic frameworks such as *spatial IC3*. Unlike traditional IC3 [34], which progresses temporally, spatial IC3 additionally partitions the circuit into spatial frames. These frames can leverage diverse representations of the transition relation—such as those produced by preprocessing techniques—to improve specialization and scalability. The algorithm also mines invariants at the frame boundaries as a side product.

Certificates with Scalable Complexity. Our certificates are currently checked in coNP [9]. However, as with SAT, they could be annotated with additional hints to reduce the complexity of certificate checking. For instance, by embedding the LRAT proof [43] produced by a SAT solver into the certificate. On the other hand, certificates can be seen as guides for solving the model checking problem itself. From this perspective, certificates merely act as hints that enable more efficient checking, even if they do not reduce the asymptotic complexity.

Certificate Shrinking. In related fields, smaller certificates have been shown to offer multiple benefits [4, 40, 149]. In addition to reduced storage requirements, they are often faster to check and more useful for downstream processing. A promising approach is to utilize SAT solver proofs to automatically identify the essential components of a certificate. These reduced certificates may, in turn, help model checkers construct even smaller certificates.

Model Checking Hammers for Interactive Theorem Provers. We aim to develop model checking *hammers* for interactive theorem provers such as Isabelle [305] and Lean [306]. These hammers would allow theorem provers to identify subgoals expressible as model checking problems and invoke an external model checker. The resulting certificate could then be lifted into the prover to close the corresponding proof.

Differential Proof. Designing complex systems is typically an incremental process, with changing requirements and optimization goals. After successfully verifying a design, even a small change requires to re-verify the entire system from scratch. Leveraging an existing proof for a design with a small delta to accelerate verification could enable significantly faster iteration during the design process.

Unified View on Assertion Mining. In our chosen formalism, model, property, and certificate are essentially the same. This unified perspective allows us to extract implied assertions directly from certificates. Assertions deemed relevant by some metric may be presented to the designer to aid in future design iterations.

Liveness Certification. From a competition perspective, we propose extending mandatory certification to the liveness track of the Hardware Model Checking Competition [20]. Achieving this will require designing a practical and trustworthy certificate format tailored to liveness properties.

Certifying Word-Level Model Checking. Our certification approach is largely agnostic to the underlying theory [13], that is, the certificate checks can be performed by any theory solver capable of describing the system. The most immediate example is the use of bit-vector solvers for word-level model checking. However, word-level model checkers employ different techniques [307], that require specialized certificate constructions. Furthermore, the lack of proof-producing SMT solvers makes this a challenging open problem.

Extension to Software Verification. We aim to generalize our certification framework to software verification [297]. Enabling certified correctness for software systems would significantly broaden the impact of our methods beyond the hardware domain.

Extension to Automated Planning. The field of automated planning [18, 308] shares strong conceptual ties with model checking. In hardware verification, system models are typically provided in semantically clear formats. In contrast, accurately describing the environment in planning is more difficult. We propose leveraging the simulation-based invariants from our certification framework to *explain* and certify environment encodings. This approach would be particularly valuable in scenarios where generative AI—such as large language models—interfaces with automated planners.

Collectively, these directions aim to enhance the trustworthiness, efficiency, and utility of symbolic verification and certification methods across a wide range of domains.

Publications

[1] Armin Biere, Katalin Fazekas, Mathias Fleury, and **Nils Froleyks**. “Clausal Congruence Closure”. In: *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*. Ed. by Supratik Chakraborty and Jie-Hong Roland Jiang. Vol. 305. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 6:1–6:25. DOI: 10.4230/LIPICS.SAT.2024.6.

[2] Armin Biere, Katalin Fazekas, Mathias Fleury, and **Nils Froleyks**. “Clausal Equivalence Sweeping”. In: *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*. Ed. by Nina Narodytska and Philipp Rümmer. IEEE, 2024, pp. 1–6. DOI: 10.34727/2024/ISBN.978-3-85448-065-5_29.

[3] Armin Biere, Katalin Fazekas, Mathias Fleury, **Nils Froleyks**, and Florian Pollitt. “Clausal Equivalence Checking”. Journal Submission. 2025.

[4] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, **Nils Froleyks**, and Florian Pollitt. “CaDiCaL 2.0”. In: *CAV (1)*. Vol. 14681. Lecture Notes in Computer Science. Springer, 2024, pp. 133–152.

[5] **Nils Froleyks** and Armin Biere. “Single Clause Assumption without Activation Literals to Speed-up IC3”. In: *Formal Methods in Computer Aided Design, FMCAD 2021*. IEEE, 2021, pp. 72–76. DOI: 10.34727/2021/isbn.978-3-85448-046-4_15.

[6] Armin Biere, **Nils Froleyks**, and Wenxi Wang. “CadiBack: Extracting Backbones with CaDiCaL”. In: *SAT*. Vol. 271. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 3:1–3:12.

[7] **Nils Froleyks**, Emily Yu, and Armin Biere. “BIG Backbones”. In: *FMCAD*. IEEE, 2023, pp. 162–167.

[8] **Nils Froleyks**, Emily Yu, and Armin Biere. “Ternary Simulation as Abstract Interpretation (Work in Progress)”. In: *MBMV 2024; 27. Workshop*. VDE. 2024, pp. 148–151.

[9] **Nils Froleyks**, Emily Yu, Armin Biere, and Keijo Heljanko. “Certifying Phase Abstraction”. In: *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*. Ed. by Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt. Vol. 14739. Lecture Notes in Computer Science. Springer, 2024, pp. 284–303. DOI: 10.1007/978-3-031-63498-7_17.

[10] **Nils Froleyks**, Emily Yu, Armin Biere, and Keijo Heljanko. "Certifying Constraints in Hardware Model Checking". Submitted. 2025.

[11] **Nils Froleyks**, Emily Yu, Mathias Preiner, Armin Biere, and Keijo Heljanko. "Introducing Certificates to the Hardware Model Checking Competition". In: *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 21-25, 2025, Proceedings*. Lecture Notes in Computer Science. Springer, 2025.

[12] Emily Yu, **Nils Froleyks**, Armin Biere, and Keijo Heljanko. "Stratified Certification for K-Induction". In: *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*. Ed. by Alberto Griggio and Neha Rungta. IEEE, 2022, pp. 59–64. DOI: 10.34727/2022/ISBN.978-3-85448-053-2_11.

[13] Emily Yu, **Nils Froleyks**, Armin Biere, and Keijo Heljanko. "Towards Compositional Hardware Model Checking Certification". In: *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*. Ed. by Alexander Nadel and Kristin Yvonne Rozier. IEEE, 2023, pp. 1–11. DOI: 10.34727/2023/ISBN.978-3-85448-060-0_12.

[14] **Nils Froleyks**, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. "SAT Competition 2020". In: *Artif. Intell.* 301 (2021), p. 103572. DOI: 10.1016/j.artint.2021.103572.

[15] Armin Biere, Mathias Fleury, **Nils Froleyks**, and Marijn J. H. Heule. "The SAT Museum". In: *Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023*. Ed. by Matti Järvisalo and Daniel Le Berre. Vol. 3545. CEUR Workshop Proceedings. CEUR-WS.org, 2023, pp. 72–87. URL: https://ceur-ws.org/Vol-3545/paper6.pdf.

[16] Florian Pollitt, Mathias Fleury, Katalin Fazekas, **Nils Froleyks**, and Armin Biere. "CaDiCaL 3.0: Proofs, Algorithms, Implied Literals, and Robust Scheduling". Submitted. 2025.

[17] **Nils Froleyks** and Tomáš Balyo. "Using an algorithm portfolio to solve sokoban". In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 8. 2017, pp. 165–166.

[18] **Nils Froleyks**, Tomáš Balyo, and Dominik Schreiber. "PASAR - Planning as Satisfiability with Abstraction Refinement". In: *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*. Ed. by Pavel Surynek and William Yeoh. AAAI Press, 2019, pp. 70–78.

[19] Tomáš Balyo and **Nils Froleyks**. "Ai assisted design of sokoban puzzles using automated planning". In: *International Conference on ArtsIT, Interactivity and Game Creation*. Springer. 2021, pp. 424–441.

Technical Reports

[20] Armin Biere, **Nils Froleyks**, and Mathias Preiner. *Hardware Model Checking Competition (HWMCC'20)*. 2020. URL: https://hwmcc.github.io/2020.

[21] Armin Biere, **Nils Froleyks**, and Mathias Preiner. "Hardware Model Checking Competition 2024". In: *Proceedings 24th International Conference on Formal Methods in Computer-Aided Design (FMCAD'24)*. Ed. by Nina Narodytska and Philipp Rümmer. TU Wien Academic Press, 2024, p. 7. DOI: 10.34727/2024/isbn.978-3-85448-065-5_6.

[22] Tomáš Balyo, **Nils Froleyks**, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, eds. *Proceedings of SAT Competition 2020; Solver and Benchmark Descriptions*. Vol. B-2020-1. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, 2020.

[23] Tomáš Balyo, **Nils Froleyks**, Markus Iser, Matti Järvisalo, and Martin Suda. "The results of SAT competition 2021". In: *Sat* 2021 (2021).

[24] **Nils Froleyks**, Emily Yu, and Armin Biere. *Unique Reconfiguration Sequence*. Tech. rep. SAT Competition, 2022.

[25] Emily Yu, **Nils Froleyks**, Armin Biere, and Mathias Fleury. *Hardware Model Checking Certificates*. Tech. rep. SAT Competition, 2022.

[26] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, **Nils Froleyks**, and Florian Pollitt. *Hardware Equivalence Checking Problems Submitted to the SAT Competition 2024*. Tech. rep. SAT Competition, 2024.

[27] **Nils Froleyks**, Emily Yu, and Armin Biere. *Challenging Certificates from Model Checking*. Tech. rep. SAT Competition, 2025.

[28] **Nils Froleyks**, Tomáš Balyo, and Dominik Schreiber. *PASAR Entering the Sparkle Planning Challenge 2019*. Tech. rep. Sparkle Planning Challenge, 2019.

[29] **Nils Froleyks**, Emily Yu, and Armin Biere. *ReconfAIGERation entering Core Challenge 2022*. Tech. rep. Core Challenge, 2022.

[30] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, **Nils Froleyks**, and Florian Pollitt. "CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024". In: *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*. Ed. by Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2024-1. Department of Computer Science Report Series B. University of Helsinki, 2024, pp. 8–10.

[31] Armin Biere, **Nils Froleyks**, and Wenxi Wang. *Sampled and Normalized Satisfiable Instances from the main track of the SAT Competition 2004 to 2022*. Mar. 2023. DOI: 10.5281/zenodo.7750076.

[32] **Nils Froleyks**. "KB3 - Keep Big Backbones". In: (2023). http://fmv.jku.at/kb3.

Bibliography

[33] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, eds. *Handbook of Model Checking*. Springer, 2018.

[34] Bradley, Aaron R. "Understanding IC3". en. In: *Theory and Applications of Satisfiability Testing – SAT 2012*. Ed. by Cimatti, Alessandro and Sebastiani, Roberto. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, 1–14. ISBN: 978-3-642-31612-8. DOI: {10.1007/978-3-642-31612-8_1}.

[35] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *FMCAD*. Vol. 1954. Lecture Notes in Computer Science. Springer, 2000, pp. 108–125.

[36] Grigorii Samuilovich Tseitin. "On the complexity of derivation in propositional calculus". In: *Studies in Mathematics and Mathematical Logic* 2 (1968), pp. 115–125.

[37] Marques-Silva JP GRASP. "A search algorithm for propositional satisfiability/JP Marques-Silva, KA Sakallah". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521.

[38] Peter Lammich. "Fast and Verified UNSAT Certificate Checking". In: *International Joint Conference on Automated Reasoning*. Springer. 2024, pp. 439–457.

[39] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. "cake_lpr: Verified Propagation Redundancy Checking in CakeML". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 223–241. DOI: 10.1007/978-3-030-72013-1_12.

[40] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. "Trimming while checking clausal proofs". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 181–188.

[41] Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. "Effective Auxiliary Variables via Structured Reencoding". In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*. Ed. by Meena Mahajan and Friedrich Slivovsky. Vol. 271. LIPIcs.

Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 11:1–11:19. DOI: 10.4230/LIPICS.SAT.2023.11.

[42] Marijn J. H. Heule. "The DRAT format and DRAT-trim checker". In: *CoRR* abs/1610.06229 (2016).

[43] Florian Pollitt, Mathias Fleury, and Armin Biere. "Faster LRAT Checking Than Solving with CaDiCaL". In: *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Ed. by Meena Mahajan and Friedrich Slivovsky. Vol. 271. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 21:1–21:12. DOI: 10.4230/LIPIcs.SAT.2023.21.

[44] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. "Strong Extension-Free Proof Systems". In: *J. Autom. Reason.* 64.3 (2020), pp. 533–554. DOI: 10.1007/S10817-019-09516-0.

[45] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. "Certified Dominance and Symmetry Breaking for Combinatorial Optimisation". In: *J. Artif. Intell. Res.* 77 (2023), pp. 1539–1589.

[46] Leszek Aleksander Kołodziejczyk and Neil Thapen. *The Strength of the Dominance Rule*. June 2024. DOI: 10.48550/arXiv.2406.13657.

[47] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded model checking using satisfiability solving". In: *Formal methods in system design* 19 (2001), pp. 7–34.

[48] Eén, Niklas and Sörensson, Niklas. "Temporal Induction by Incremental SAT Solving". en. In: *Electronic Notes in Theoretical Computer Science*. BMC'2003, First International Workshop on Bounded Model Checking 89.4 (Jan. 2003), 543–560. ISSN: 1571-0661. DOI: {10.1016/S1571-0661(05)82542-3}.

[49] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. Ed. by Ranjit Jhala and David A. Schmidt. Vol. 6538. Lecture Notes in Computer Science. Springer, 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[50] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Tech. rep. 07/1. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, 2007.

[51] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 and Beyond*. Tech. rep. 11/2. Institute for Formal Models and Verification, Johannes Kepler University, 2011.

[52] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *STOC*. 1971, pp. 151–158.

[53] B. A. Trakhtenbrot. “A Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms”. In: *Annals of the History of Computing* 6.4 (Oct. 1984), pp. 384–400. ISSN: 0164-1239. DOI: 10.1109/MAHC.1984.10036.

[54] L. J. Stockmeyer and A. R. Meyer. “Word Problems Requiring Exponential Time(Preliminary Report)”. In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC ’73. New York, NY, USA: Association for Computing Machinery, Apr. 1973, pp. 1–9. ISBN: 978-1-4503-7430-9. DOI: 10.1145/800125.804029.

[55] Walter J. Savitch. “Relationships between Nondeterministic and Deterministic Tape Complexities”. In: *Journal of Computer and System Sciences* 4.2 (Apr. 1970), pp. 177–192. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(70)80006-X.

[56] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. ISBN: 978-0-521-42426-4. URL: http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264.

[57] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. “Robust Boolean reasoning for equivalence checking and functional property verification”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 21.12 (2002), pp. 1377–1394. DOI: 10.1109/TCAD.2002.804386.

[58] Tomáš Balyo, Armin Biere, Markus Iser, and Carsten Sinz. “SAT Race 2015”. In: *Artif. Intell.* 241 (2016), pp. 45–65.

[59] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. “GANAK: A Scalable Probabilistic Exact Model Counter.” In: *IJCAI*. Vol. 19. 2019, pp. 1169–1176.

[60] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. “Preprocessing in SAT Solving”. In: Frontiers in Artificial Intelligence and Applications 336 (2021). Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, pp. 391–435. DOI: 10.3233/FAIA200992.

[61] Matti Järvisalo, Marijn Heule, and Armin Biere. “Inprocessing Rules”. In: *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Vol. 7364. Lecture Notes in Computer Science. Springer, 2012, pp. 355–370. DOI: 10.1007/978-3-642-31365-3_28.

[62] Katalin Fazekas, Armin Biere, and Christoph Scholl. “Incremental Inprocessing in SAT Solving”. In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikoláš Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 136–154. DOI: 10.1007/978-3-030-24258-9_9.

[63] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. “SATIRE: A New Incremental Satisfiability Engine”. In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001, pp. 542–545. DOI: 10.1145/378239.379019.

[64] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. “Preprocessing in Incremental SAT”. In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 256–269. DOI: 10.1007/978-3-642-31612-8_20.

[65] Alexander Nadel and Vadim Ryvchin. “Efficient SAT Solving under Assumptions”. In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 242–255. DOI: 10.1007/978-3-642-31612-8_19.

[66] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. “Ultimately Incremental SAT”. In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Carsten Sinz and Uwe Egly. Vol. 8561. Lecture Notes in Computer Science. Springer, 2014, pp. 206–218. DOI: 10.1007/978-3-319-09284-3_16.

[67] Stefan Kupferschmid, Matthew Lewis, Tobias Schubert, and Bernd Becker. “Incremental preprocessing methods for use in BMC”. In: *Formal Methods Syst. Des.* 39.2 (2011), pp. 185–204. DOI: 10.1007/S10703-011-0122-4.

[68] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

[69] Armin Biere. “CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017”. In: *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Marijn J. H. Heule, and Matti Järvisalo. Vol. B-2017-1. Department of Computer Science Series of Publications B. University of Helsinki, 2017, pp. 14–15.

[70] Biere, Armin. “Cadical, Lingeling, Plingeling, Treengeling and Yalsat Entering the Sat Competition 2018”. In: 2017, 14–15.

[71] Armin Biere. “CaDiCaL at the SAT Race 2019”. In: *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Ed. by Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. Vol. B-2019-1. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, 2019, pp. 8–9.

[72] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020”. In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[73] Armin Biere, Mathias Fleury, and Maximillian Heisinger. “CaDiCaL, Kissat, Paracooba Entering the SAT Competition 2021”. In: *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2021-1. Department of Computer Science Report Series B. University of Helsinki, 2021, pp. 10–13.

[74] Armin Biere, Mathias Fleury, and Florian Pollitt. “CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT Entering the SAT Competition 2023”. In: *Proc. of SAT Competition 2023 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2023-1. Department of Computer Science Report Series B. University of Helsinki, 2023, pp. 14–15.

[75] Mate Soos, Karsten Nohl, and Claude Castelluccia. “Extending SAT Solvers to Cryptographic Problems”. In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24.

[76] Gilles Audemard and Laurent Simon. “On the Glucose SAT Solver”. In: *Int. J. Artif. Intell. Tools* 27.1 (2018), 1840001:1–1840001:25. DOI: 10.1142/S0218213018400018.

[77] Alexander Nadel. “Introducing Intel(R) SAT Solver”. In: *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Ed. by Kuldeep S. Meel and Ofer Strichman. Vol. 236. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 8:1–8:23. ISBN: 978-3-95977-242-6. DOI: 10.4230/LIPIcs.SAT.2022.8. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16682.

[78] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. “Automated Reencoding of Boolean Formulas”. In: *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*. Ed. by Armin Biere, Amir Nahir, and Tanja E. J. Vos. Vol. 7857. Lecture Notes in Computer Science. Springer, 2012, pp. 102–117. DOI: 10.1007/978-3-642-39611-3_14.

[79] Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stenfa Szeider, and Armin Biere. “IPASIR-UP: User Propagators for CDCL”. In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, Alghero, Italy*. Ed. by Meena Mahajan and Friedrich Slivovsky. Vol. 271. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 8:1–8:13. DOI: 10.4230/LIPICS.SAT.2023.8.

[80] Emre Yolcu, Scott Aaronson, and Marijn J. H. Heule. “An Automated Approach to the Collatz Conjecture”. In: *J. Autom. Reason.* 67.2 (2023), p. 15. DOI: 10.1007/S10817-022-09658-8.

[81] Bernardo Subercaseaux and Marijn J. H. Heule. “The Packing Chromatic Number of the Infinite Square Grid is 15”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13993. Lecture Notes in Computer Science. Springer, 2023, pp. 389–406. DOI: 10.1007/978-3-031-30823-9_20.

[82] David Neiman, John Mackey, and Marijn J. H. Heule. “Tighter Bounds on Directed Ramsey Number R(7)”. In: *Graphs Comb.* 38.5 (2022), p. 156. DOI: 10.1007/S00373-022-02560-5.

[83] Evan Lohn, Chris Lambert, and Marijn J. H. Heule. “Compact Symmetry Breaking for Tournaments”. In: *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*. Ed. by Alberto Griggio and Neha Rungta. IEEE, 2022, pp. 179–188. DOI: 10.34727/2022/ISBN.978-3-85448-053-2_24.

[84] Alexey Ignatiev, António Morgado, and João Marques-Silva. “PySAT: A Python Toolkit for Prototyping with SAT Oracles”. In: *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Vol. 10929. Lecture Notes in Computer Science. Springer, 2018, pp. 428–437. DOI: 10.1007/978-3-319-94144-8_26.

[85] Aina Niemetz, Mathias Preiner, and Armin Biere. “Boolector at the SMT competition 2019”. In: *Proceedings of the 17th International Workshop on Satisfiability Modulo Theories (SMT 2019), affiliated with the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT 2019), Lisbon,*

Portugal, July 7-8, 2019. Ed. by Joe Hendrix and Natasha Sharygina. 2019, 2 pages.

[86] Aina Niemetz and Mathias Preiner. “Bitwuzla”. In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*. Ed. by Constantin Enea and Akash Lal. Vol. 13965. Lecture Notes in Computer Science. Springer, 2023, pp. 3–17. DOI: 10.1007/978-3-031-37703-7_1.

[87] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. “cvc5: A Versatile and Industrial-Strength SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[88] Fahiem Bacchus. “MaxHS in the 2022 MaxSat Evaluation”. In: *Proc. of MaxSAT Evaluation 2020 – Solver and Benchmark Descriptions*. Ed. by Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins. Vol. B-2022-2. Department of Computer Science Series of Publications B. University of Helsinki, 2022, p. 17.

[89] Peter Sanders and Dominik Schreiber. “Mallob: Scalable SAT Solving On Demand With Decentralized Job Scheduling”. In: *J. Open Source Softw.* 7.77 (2022), p. 4591. DOI: 10.21105/JOSS.04591.

[90] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. “A Flexible Proof Format for SAT Solver-Elaborator Communication”. In: *Log. Methods Comput. Sci.* 18.2 (2022). DOI: 10.46298/lmcs-18(2:3)2022.

[91] Nick Feng and Fahiem Bacchus. “Clause Size Reduction with all-UIP Learning”. In: *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*. Ed. by Luca Pulina and Martina Seidl. Vol. 12178. Lecture Notes in Computer Science. Springer, 2020, pp. 28–45. DOI: 10.1007/978-3-030-51825-7_3.

[92] Benjamin Kiesl-Reiter and Michael W. Whalen. “Proofs for Incremental SAT with Inprocessing”. In: *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*. Ed. by Alexander Nadel and Kristin Yvonne Rozier. IEEE, 2023, pp. 132–140. DOI: 10.34727/2023/ISBN.978-3-85448-060-0_21.

[93] Armin Biere, Md. Solimul Chowdhury, Marijn J. H. Heule, Benjamin Kiesl, and Michael W. Whalen. “Migrating Solver State”. In: *SAT*. Vol. 236. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 27:1–27:24. DOI: 10.4230/LIPICS.SAT.2022.27.

[94] Randy Hickey and Fahiem Bacchus. “Trail Saving on Backtrack”. In: *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*. Ed. by Luca Pulina and Martina Seidl. Vol. 12178. Lecture Notes in Computer Science. Springer, 2020, pp. 46–61.

[95] Mathias Fleury and Armin Biere. “Efficient All-UIP Learned Clause Minimization”. In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu-Min Li and Felip Manyà. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 171–187. DOI: 10.1007/978-3-030-80223-3_12.

[96] Mathias Fleury and Peter Lammich. “A More Pragmatic CDCL for IsaSAT and Targetting LLVM (Short Paper)”. In: *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*. Ed. by Brigitte Pientka and Cesare Tinelli. Vol. 14132. Lecture Notes in Computer Science. Springer, 2023, pp. 207–219. DOI: 10.1007/978-3-031-38499-8_12.

[97] Daniel Le Berre, Olivier Roussel, and Laurent Simon. *SAT Competition 2009: Benchmark Submission Guidelines*. https://web.archive.org/web/20190325181937/https://www.satcompetition.org/2009/format-benchmarks2009.html. Accessed: 2024-01-15. 2009.

[98] Gilles Audemard and Laurent Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers”. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 399–404.

[99] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. “Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling”. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 463–484. DOI: 10.1007/978-3-030-53288-8_22.

[100] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. “Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction”. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*. Ed. by Matti Järvisalo and Allen Van Gelder. Vol. 7962. Lecture Notes in Computer Science. Springer, 2013, pp. 309–317. DOI: 10.1007/978-3-642-39071-5_23.

[101] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. “DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs”. In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Carsten Sinz and Uwe Egly. Vol. 8561. Lecture Notes in Computer Science. Springer, 2014, pp. 422–429. DOI: 10.1007/978-3-319-09284-3_31.

[102] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. “Efficient Certified RAT Verification”. In: *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*. Ed. by Leonardo de Moura. Vol. 10395. Lecture Notes in Computer Science. Springer, 2017, pp. 220–236. DOI: 10.1007/978-3-319-63046-5_14.

[103] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. “Certified Symmetry and Dominance Breaking for Combinatorial Optimisation”. In: *Journal of Artificial Intelligence Research* 77 (Aug. 2023). Preliminary version in *AAAI ’22*, pp. 1539–1589.

[104] Matti Järvisalo and Armin Biere. “Reconstructing Solutions after Blocked Clause Elimination”. In: *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Ofer Strichman and Stefan Szeider. Vol. 6175. Lecture Notes in Computer Science. Springer, 2010, pp. 340–345. DOI: 10.1007/978-3-642-14186-7_30.

[105] Niklas Eén and Armin Biere. “Effective Preprocessing in SAT Through Variable and Clause Elimination”. In: *SAT*. Vol. 3569. Lecture Notes in Computer Science. Springer, 2005, pp. 61–75.

[106] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. “Vivifying Propositional Clausal Formulae”. In: *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*. Ed. by Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris. Vol. 178. Frontiers in Artificial Intelligence and Applications. IOS Press, 2008, pp. 525–529. DOI: 10.3233/978-1-58603-891-5-525.

[107] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. “An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers”. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. Ed. by Carles Sierra. ijcai.org, 2017, pp. 703–711. DOI: 10.24963/IJCAI.2017/98.

[108] Chu Min Li. “Integrating Equivalency Reasoning into Davis-Putnam Procedure”. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*. Ed. by Henry A. Kautz and Bruce W. Porter. AAAI Press / The MIT Press, 2000, pp. 291–296.

[109] Gunnar Andersson, Per Bjesse, Byron Cook, and Ziyad Hanna. “A proof engine approach to solving combinational design automation problems”. In: *Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002*. ACM, 2002, pp. 725–730. DOI: 10.1145/513918.514101.

[110] Marijn Heule, Matti Järvisalo, and Armin Biere. “Revisiting Hyper Binary Resolution”. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*. Ed. by Carla P. Gomes and Meinolf Sellmann. Vol. 7874. Lecture Notes in Computer Science. Springer, 2013, pp. 77–93. DOI: 10.1007/978-3-642-38171-3_6.

[111] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. “Covered Clause Elimination”. In: *Short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR-17-short, Yogyakarta, Indonesia, October 10-15, 2010*. Ed. by Andrei Voronkov, Geoff Sutcliffe, Matthias Baaz, and Christian G. Fermüller. Vol. 13. EPiC Series in Computing. EasyChair, 2010, pp. 41–46. DOI: 10.29007/CL8S.

[112] Lee A. Barnett, David M. Cerna, and Armin Biere. “Covered Clauses Are Not Propagation Redundant”. In: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12166. Lecture Notes in Computer Science. Springer, 2020, pp. 32–47. DOI: 10.1007/978-3-030-51074-9_3.

[113] Matti Järvisalo, Armin Biere, and Marijn J. H. Heule. “Blocked Clause Elimination”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by Javier Esparza and Rupak Majumdar. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 129–144. DOI: 10.1007/978-3-642-12002-2_10.

[114] Shaowei Cai, Xindi Zhang, Mathias Fleury, and Armin Biere. “Better Decision Heuristics in CDCL through Local Search and Target Phases”. In: *J. Artif. Intell. Res.* 74 (2022), pp. 1515–1563. DOI: 10.1613/jair.1.13666.

[115] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. “Clause Elimination Procedures for CNF Formulas”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*. Ed. by Christian G. Fermüller and Andrei Voronkov. Vol. 6397. Lecture Notes in Computer Science. Springer, 2010, pp. 357–371. DOI: 10.1007/978-3-642-16242-8_26.

[116] Benjamin Kiesl, Marijn J. H. Heule, and Armin Biere. “Truth Assignments as Conditional Autarkies”. In: *Automated Technology for Verification and Analysis – 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Vol. 11781. Lecture Notes in Computer Science. Springer, 2019, pp. 48–64. DOI: 10.1007/978-3-030-31784-3_3.

[117] Alexander Nadel and Vadim Ryvchin. “Chronological Backtracking”. In: *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Vol. 10929. Lecture Notes in Computer Science. Springer, 2018, pp. 111–121. DOI: 10.1007/978-3-319-94144-8_7.

[118] Sibylle Möhle and Armin Biere. “Backing Backtracking”. In: *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Ed. by Mikolás Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Springer, 2019, pp. 250–266. DOI: 10.1007/978-3-030-24258-9_18.

[119] Knot Pipatsrisawat and Adnan Darwiche. “A Lightweight Component Caching Scheme for Satisfiability Solvers”. In: *Theory and Applications of Satisfiability Testing–SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*. Ed. by João Marques-Silva and Karem A. Sakallah. Vol. 4501. Lecture Notes in Computer Science. Springer, 2007, pp. 294–299. DOI: 10.1007/978-3-540-72788-0_28.

[120] Peter van der Tak, Antonio Ramos, and Marijn J. H. Heule. “Reusing the Assignment Trail in CDCL Solvers”. In: *J. Satisf. Boolean Model. Comput.* 7.4 (2011), pp. 133–138. DOI: 10.3233/SAT190082. URL: https://doi.org/10.3233/sat190082.

[121] HyoJung Han and Fabio Somenzi. “On-the-Fly Clause Improvement”. In: *Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30–July 3, 2009. Proceedings*. Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 209–222. DOI: 10.1007/978-3-642-02777-2_21.

[122] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. “Learning for Dynamic Subsumption”. In: *ICTAI 2009, 21st IEEE International Conference on Tools with Artificial Intelligence, Newark, New Jersey, USA, 2-4 November 2009*. IEEE Computer Society, 2009, pp. 328–335. DOI: 10.1109/ICTAI.2009.22.

[123] Lintao Zhang. “On Subsumption Removal and On-the-Fly CNF Simplification”. In: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*. Ed. by Fahiem Bacchus and Toby Walsh. Vol. 3569. Lecture Notes in Computer Science. Springer, 2005, pp. 482–489. DOI: 10.1007/11499107_42.

[124] Siert Wieringa, Matti Niemenmaa, and Keijo Heljanko. “Tarmo: A Framework for Parallelized Bounded Model Checking”. In: *Proceedings 8th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009*. Ed. by Lubos Brim and Jaco van de Pol. Vol. 14. EPTCS. 2009, pp. 62–76. DOI: 10.4204/EPTCS.14.5.

[125] Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. “Satisfiability Modulo Custom Theories in Z3”. In: *VMCAI*. Vol. 13881. Lecture Notes in Computer Science. Springer, 2023, pp. 91–105.

[126] Vijay Ganesh, Charles W. O’Donnell, Mate Soos, Srinivas Devadas, Martin C. Rinard, and Armando Solar-Lezama. “Lynx: A Programmatic SAT Solver for the RNA-Folding Problem”. In: *SAT*. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 143–156.

[127] João P. Marques Silva and Karem A. Sakallah. “GRASP - a new search algorithm for satisfiability”. In: *ICCAD*. 1996, pp. 220–227.

[128] João Marques-Silva, Inês Lynce, and Sharad Malik. “Conflict-Driven Clause Learning SAT Solvers”. In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 133–182. DOI: 10.3233/FAIA200987.

[129] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3. URL: https://doi.org/10.1007/978-3-540-78800-3.

[130] Bruno Dutertre. “Yices 2.2”. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[131] Tianwei Zhang and Stefan Szeider. “Searching for Smallest Universal Graphs and Tournaments with SAT”. In: *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*. Ed. by Roland H. C. Yap. Vol. 280. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 39:1–39:20. DOI: 10.4230/LIPICS.CP.2023.39.

[132] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. “Theory Solving Made Easy with Clingo 5”. In: *ICLP (Technical Communications)*. Vol. 52. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 2:1–2:15.

[133] Marijn J. H. Heule and Armin Biere. “Proofs for Satisfiability Problems”. In: *All about Proofs, Proofs for All (APPA)*. Vol. 55. Math. Logic and Foundations. College Pub., 2015.

[134] Marijn J. H. Heule. “Proofs of Unsatisfiability”. In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 635–668. DOI: 10.3233/FAIA200998.

[135] Eugene Goldberg and Yakov Novikov. “Verification of proofs of unsatisfiability for CNF formulas”. In: *2003 Design, Automation and Test in Europe Conference and Exhibition* (2003), pp. 886–891. URL: https://api.semanticscholar.org/CorpusID:10504432.

[136] Tomáš Balyo and Marijn J. H. Heule. “Proceedings of SAT Competition 2016 – Solver and Benchmark Descriptions”. In: ed. by Matti Järvisalo. Vol. B-2016-1. Department of Computer Science Series of Publications B. University of Helsinki, 2016.

[137] Tomáš Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, eds. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. English. Department of Computer Science Series of Publications B. Finland: Department of Computer Science, University of Helsinki, 2023.

[138] Peter Lammich. “Efficient Verified (UN)SAT Certificate Checking”. In: *J. Autom. Reason.* 64.3 (2020), pp. 513–532.

[139] Stephan Gocht and Jakob Nordström. “Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs”. In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI ’21)*. Feb. 2021, pp. 3768–3777.

[140] Stephan Gocht. “Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning”. Available at https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu. PhD thesis. Lund, Sweden: Lund University, June 2022.

[141] Tobias Paxian, Sven Reimer, and Bernd Becker. “Dynamic Polynomial Watchdog Encoding for Solving Weighted MaxSAT”. In: *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Vol. 10929. Lecture Notes in Computer Science. Springer, 2018, pp. 37–53. DOI: 10.1007/978-3-319-94144-8_3.

[142] Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. “Incremental Proofs for Bounded Model Checking”. In: *Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems, MBMV 2024, Kaiserslautern, Germany, February 14-15, 2023*. Ed. by Wolfgang Kunz and Daniel Große. ITG Fachberichte. Accepted. VDE Verlag, 2024.

[143] Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. “Certifying Incremental SAT Solving”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 25th International Conference, LPAR-25, Balaclava, Mauritius, 26-31 May, 2024. Proceedings*. Ed. by Nikolaj Bjorner, Marijn Heule, and Andrei Voronkov. To appear. 2024.

[144] Armin Biere and Mathias Fleury. “Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022”. In: *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2022-1. Department of Computer Science Series of Publications B. University of Helsinki, 2022, pp. 10–11.

[145] Mikoláš Janota, Inês Lynce, and João Marques-Silva. “Algorithms for computing backbones of propositional formulae”. In: *AI Commun.* 28.2 (2015), pp. 161–177. DOI: 10.3233/AIC-140640.

[146] Anton Belov and João Marques-Silva. “Accelerating MUS extraction with recursive model rotation”. In: *International Conference on Formal Methods in Computer-Aided Design, FMCAD ’11, Austin, TX, USA, October 30 - November 02, 2011*. Ed. by Per Bjesse and Anna Slobodová. FMCAD Inc., 2011, pp. 37–40.

[147] Tobias Faller, Nikolaos Ioannis Deligiannis, Markus Schwörer, Matteo Sonza Reorda, and Bernd Becker. “Constraint-Based Automatic SBST Generation for RISC-V Processor Families”. In: *IEEE European Test Symposium, ETS 2023, Venezia, Italy, May 22-26, 2023*. IEEE, 2023, pp. 1–6. DOI: 10.1109/ETS56758.2023.10174156.

[148] Arie Gurfinkel and Yakir Vizel. “DRUPing for interpolates”. In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 2014, pp. 99–106. DOI: 10.1109/FMCAD.2014.6987601.

[149] Stefan Kupferschmid. “Über Craigsche Interpolation und deren Anwendung in der formalen Modellprüfung”. PhD thesis. University of Freiburg, 2013. ISBN: 978-3-86247-411-0.

[150] Cyrille Artho, Armin Biere, and Martina Seidl. “Model-Based Testing for Verification Back-Ends”. In: *TAP @STAF*. Vol. 7942. Lecture Notes in Computer Science. Springer, 2013, pp. 39–55.

[151] Aina Niemetz, Mathias Preiner, and Armin Biere. “Model-Based API Testing for SMT Solvers”. In: *SMT*. Vol. 1889. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 3–14.

[152] Aina Niemetz, Mathias Preiner, and Clark W. Barrett. “Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers”. In: *CAV (2)*. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 92–106.

[153] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. “StringFuzz: A Fuzzer for String Solvers”. In: *CAV (2)*. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 45–51.

[154] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Generative type-aware mutation for testing SMT solvers". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–19.

[155] Joseph Scott, Trishal Sudula, Hammad Rehman, Federico Mora, and Vijay Ganesh. "BanditFuzz: Fuzzing SMT Solvers with Multi-agent Reinforcement Learning". In: *FM*. Vol. 13047. Lecture Notes in Computer Science. Springer, 2021, pp. 103–121.

[156] Mauro Bringolf, Dominik Winterer, and Zhendong Su. "Finding and Understanding Incompleteness Bugs in SMT Solvers". In: *ASE*. ACM, 2022, 43:1–43:10.

[157] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. "Detecting critical bugs in SMT solvers using blackbox mutational fuzzing". In: *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 701–712.

[158] Robert Brummayer, Florian Lonsing, and Armin Biere. "Automated Testing and Debugging of SAT and QBF Solvers". In: *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Ofer Strichman and Stefan Szeider. Vol. 6175. Lecture Notes in Computer Science. Springer, 2010, pp. 44–57. DOI: 10.1007/978-3-642-14186-7_6.

[159] Andrew Haberlandt and Harrison Green. "SBVA-CADICAL and SBVA-KISSAT: Structured Bounded Variable Addition". In: *Proc. of SAT Competition 2023 – Solver and Benchmark Descriptions*. Ed. by Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2023-1. Department of Computer Science Report Series B. University of Helsinki, 2023, p. 18.

[160] Armin Biere, Tom van Dijk, and Keijo Heljanko. "Hardware Model Checking Competition 2017". In: *Formal Methods in Computer-Aided Design, FMCAD*. Ed. by Daryl Stewart and Georg Weissenbacher. IEEE, 2017, p. 9.

[161] Jingchao Chen. "optsat, abcdsat and Solvers Based on Simplified Data Structure and Hybrid Solving Strategies". In: *Proceedings of SAT Competition 2020: Solver and benchmark descriptions* (2020), p. 25.

[162] Mate Soos, Bart Selman, Henry Kautz, Jo Devriendt, and Stephan Gocht. "CryptoMiniSat with WalkSAT at the SAT Competition 2020". In: *Proceedings of SAT Competition 2020: Solver and benchmark descriptions* (2020), p. 29.

[163] Mate Soos, Jo Devriendt, Stephan Gocht, Arijit Shaw, and Kuldeep S Meel. "CryptoMiniSat with ccanr at the SAT Competition 2020". In: vol. 2020. 2020, p. 27.

[164] Norbert Manthey. "Riss 7 in Proceedings of SAT Competition 2020". In: *Proceedings of SAT Competition 2020: Solver and benchmark descriptions* (2020).

[165] Stepan Kochemazov, Alexey Ignatiev, and João Marques-Silva. "Assessing Progress in SAT Solvers Through the Lens of Incremental SAT". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*. Ed. by Chu-Min Li and Felip Manyà. Vol. 12831. Lecture Notes in Computer Science. Springer, 2021, pp. 280–298. DOI: 10.1007/978-3-030-80223-3_20.

[166] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs". In: *Proc. of TACAS'99*. Vol. 1579. LNCS. Springer, 1999, pp. 193–207.

[167] Bjesse, Per and Claessen, Koen. "SAT-Based Verification without State Space Traversal". en. In: *Formal Methods in Computer-Aided Design*. Ed. by Hunt, Warren A. and Johnson, Steven D. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, 409–426. ISBN: 978-3-540-40922-9. DOI: {10.1007/3-540-40922-X_23}.

[168] Cabodi, G. and Camurati, P. E. and Mishchenko, A. and Palena, M. and Pasini, P. "SAT Solver Management Strategies in IC3: An Experimental Approach". en. In: *Formal Methods in System Design* 50 (Mar. 2017), 39–74. ISSN: 1572-8102. DOI: {10.1007/s10703-017-0272-0}.

[169] Barrett, Clark and Stump, Aaron and Tinelli, Cesare and others. "The Smt-Lib Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

[170] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. "Zchaff 2004: An Efficient SAT Solver". In: *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*. Ed. by Holger H. Hoos and David G. Mitchell. Vol. 3542. Lecture Notes in Computer Science. Springer, 2004, pp. 360–375. DOI: 10.1007/11527695_27.

[171] Preiner, Mathias and Biere, Armin. *Hardware Model Checking Competition 2019*. http://fmv.jku.at/hwmcc19/. 2019.

[172] Van Gelder, Allen. "Autarky Pruning in Propositional Model Elimination Reduces Failure Redundancy". en. In: *Journal of Automated Reasoning* 23.2 (Aug. 1999), 137–193. ISSN: 1573-0670. DOI: {10.1023/A:1006143621319}.

[173] Evguenii I. Goldberg and Yakov Novikov. "BerkMin: A Fast and Robust SAT-Solver". In: *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*. IEEE Computer Society, 2002, pp. 142–149. DOI: 10.1109/DATE.2002.998262.

[174] Gershman, Roman and Strichman, Ofer. "HaifaSat: A New Robust SAT Solver". In: *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*. Ed. by Ur, Shmuel and Bin, Eyal and Wolfsthal, Yaron. Vol. 3875.

Lecture Notes in Computer Science. Springer, 2005, 76–89. DOI: {10.1007/11678779_6}.

[175] Hickey, Randy and Bacchus, Fahiem. “Speeding Up Assumption-Based SAT”. en. In: *Theory and Applications of Satisfiability Testing – SAT 2019*. Ed. by Janota, Mikoláš and Lynce, Inês. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, 164–182. ISBN: 978-3-030-24258-9. DOI: {10.1007/978-3-030-24258-9_11}.

[176] Lagniez, Jean-Marie and Biere, Armin. “Factoring Out Assumptions to Speed Up MUS Extraction”. en. In: *Theory and Applications of Satisfiability Testing – SAT 2013*. Ed. by Järvisalo, Matti and Van Gelder, Allen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, 276–292. ISBN: 978-3-642-39071-5. DOI: {10.1007/978-3-642-39071-5_21}.

[177] Robert K. Brayton and Alan Mishchenko. “ABC: An Academic Industrial-Strength Verification Tool”. In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul B. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 24–40.

[178] Biere, Armin and Järvisalo, Matti and Le Berre, Daniel and Meel, Kuldeep S. and Mengel, Stefan. *The SAT Practitioner’s Manifesto*. eng. Sept. 2020. DOI: {10.5281/zenodo.4500928}.

[179] Vizel, Y. and Grumberg, O. and Shoham, S. “Lazy Abstraction and SAT-Based Reachability in Hardware Model Checking”. In: *2012 Formal Methods in Computer-Aided Design (FMCAD)*. Oct. 2012, 173–181.

[180] Andrew J. Parkes. “Clustering at the Phase Transition”. In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*. Ed. by Benjamin Kuipers and Bonnie L. Webber. AAAI Press / The MIT Press, 1997, pp. 340–345.

[181] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. “Determining Computational Complexity from Characteristic ’Phase Transitions’”. In: *Nature* 400 (July 1999), pp. 133–137. DOI: 10.1038/22055.

[182] Ryan Williams, Carla P. Gomes, and Bart Selman. “Backdoors To Typical Case Complexity”. In: *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*. Ed. by Georg Gottlob and Toby Walsh. Morgan Kaufmann, 2003, pp. 1173–1178.

[183] Michael Codish, Yoav Fekete, and Amit Metodi. “Backbones for Equality”. In: *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*. Ed. by Valeria Bertacco and Axel Legay. Vol. 8244. Lecture Notes in Computer Science. Springer, 2013, pp. 1–14. DOI: 10.1007/978-3-319-03077-7_1.

[184] Weixiong Zhang. “Phase transitions and backbones of the asymmetric traveling salesman problem”. In: *Journal of Artificial Intelligence Research* 21 (2004), pp. 471–497.

[185] Mikoláš Janota. “SAT solving in interactive configuration”. PhD thesis. University College Dublin, 2010.

[186] Philip Kilby, John Slaney, Sylvie Thiébaux, Toby Walsh, et al. “Backbones and backdoors in satisfiability”. In: *AAAI*. Vol. 5. 2005, pp. 1368–1373.

[187] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. “Post-Silicon Fault Localisation Using Maximum Satisfiability and Backbones”. In: *2011 Formal Methods in Computer-Aided Design (FMCAD)*. Oct. 2011, pp. 63–66.

[188] Charlie Shucheng Zhu, Georg Weissenbacher, Divjyot Sethi, and Sharad Malik. “SAT-based techniques for determining backbones for post-silicon fault localisation”. In: *2011 IEEE International High Level Design Validation and Test Workshop, HLDVT 2011, Napa Valley, CA, USA, November 9-11, 2011*. Ed. by Zeljko Zilic and Sandeep K. Shukla. IEEE Computer Society, 2011, pp. 84–91. DOI: 10.1109/HLDVT.2011.6113981.

[189] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. “Silicon fault diagnosis using sequence interpolation with backbones”. In: *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*. Ed. by Yao-Wen Chang. IEEE, 2014, pp. 348–355. DOI: 10.1109/ICCAD.2014.7001373.

[190] Mohamed El Bachir Menaï. “A Two-Phase Backbone-Based Search Heuristic for Partial MAX-SAT – An Initial Investigation”. In: *Innovations in Applied Artificial Intelligence*. Ed. by Moonis Ali and Floriana Esposito. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 681–684. ISBN: 978-3-540-31893-4. DOI: 10.1007/11504894_94.

[191] Guoqiang Zeng, Chongwei Zheng, Zhengjiang Zhang, and Yongzai Lu. “An Backbone Guided Extremal Optimization Method for Solving the Hard Maximum Satisfiability Problem”. In: *2012 International Conference on Computer Application and System Modeling*. Atlantis Press, Aug. 2012, pp. 1301–1304. ISBN: 978-94-91216-00-8. DOI: 10.2991/iccasm.2012.332.

[192] Weixiong Zhang. “Configuration Landscape Analysis and Backbone Guided Local Search.: Part I: Satisfiability and Maximum Satisfiability”. In: *Artificial Intelligence* 158.1 (Sept. 2004), pp. 1–26. ISSN: 0004-3702. DOI: 10.1016/j.artint.2004.04.001.

[193] Weixiong Zhang, Ananda Rangan, Moshe Looks, et al. “Backbone guided local search for maximum satisfiability”. In: *IJCAI*. Vol. 3. 2003, pp. 1179–1186.

[194] Olivier Dubois and Gilles Dequen. “A backbone-search heuristic for efficient solving of hard 3-SAT formulae”. In: *IJCAI*. Vol. 1. 2001, pp. 248–253.

[195] Miroslav N Velev. “Formal verification of VLIW microprocessors with speculative execution”. In: *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer. 2000, pp. 296–311.

[196] Andreas Kaiser and Wolfgang Küchlin. “Detecting inadmissible and necessary variables in large propositional formulae”. In: *Intl. Joint Conf. on Automated Reasoning (Short Papers)*. University of Siena. 2001, pp. 96–102.

[197] Sharlee Climer and Weixiong Zhang. “Searching for backbones and fat: A limit-crossing approach with applications”. In: *AAAI/IAAI*. 2002, pp. 707–712.

[198] João Marques-Silva, Mikoláš Janota, and Inês Lynce. “On Computing Backbones of Propositional Theories”. In: *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*. Ed. by Helder Coelho, Rudi Studer, and Michael J. Wooldridge. Vol. 215. Frontiers in Artificial Intelligence and Applications. IOS Press, 2010, pp. 15–20.

[199] Alessandro Previti and Matti Järvisalo. “A preference-based approach to backbone computation with application to argumentation”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 896–902.

[200] Yueling Zhang, Min Zhang, and Geguang Pu. “Optimizing backbone filtering”. In: *Science of Computer Programming* 187 (2020), p. 102374.

[201] Yueling Zhang, Min Zhang, Geguang Pu, Fu Song, and Jianwen Li. “Towards backbone computing: A Greedy-Whitening based approach”. In: *AI Communications* 31.3 (2018), pp. 267–280.

[202] Armin Biere. “Adaptive Restart Strategies for Conflict Driven SAT Solvers”. In: *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*. Ed. by Hans Kleine Büning and Xishun Zhao. Vol. 4996. Lecture Notes in Computer Science. Springer, 2008, pp. 28–33. DOI: 10.1007/978-3-540-79719-7_4.

[203] Armin Biere and Andreas Fröhlich. “Evaluating CDCL Variable Scoring Schemes”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*. Ed. by Marijn Heule and Sean A. Weaver. Vol. 9340. Lecture Notes in Computer Science. Springer, 2015, pp. 405–422. DOI: 10.1007/978-3-319-24318-4_29.

[204] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. “Software Fault Interactions and Implications for Software Testing”. In: *IEEE Trans. Software Eng.* 30.6 (2004), pp. 418–421. DOI: 10.1109/TSE.2004.24.

[205] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. “Where the Really Hard Problems Are”. In: *IJCAI*. 1991, pp. 331–340.

[206] Dimitris Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. “Generating Satisfiable Problem Instances”. In: *AAAI/IAAI* 2000 (2000), pp. 256–261.

[207] Tasniem Al-Yahya, Mohamed El Bachir Abdelkrim Menai, and Hassan Mathkour. “Boosting the Performance of CDCL-Based SAT Solvers by Exploiting Backbones and Backdoors”. In: *Algorithms* 15.9 (2022), p. 302.

[208] Armin Biere Mathias Fleury. “Gimsatul, IsaSAT, Kissat”. In: *SAT COMPETITION 2022* (2022), p. 10.

[209] John S. Schlipf, Fred S. Annexstein, John V. Franco, and Ramjee P. Swaminathan. “On Finding Solutions for Extended Horn Formulas”. In: *Inf. Process. Lett.* 54.3 (1995), pp. 133–137. DOI: 10.1016/0020-0190(95)00019-9.

[210] Alvaro del Val. “On 2-SAT and Renamable Horn”. In: (2000), pp. 279–284.

[211] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (1960), pp. 201–215.

[212] I. P. Gent. “Optimal Implementation of Watched Literals and More General Techniques”. In: *Journal of Artificial Intelligence Research* 48 (Oct. 2013), pp. 231–252. ISSN: 1076-9757. DOI: 10.1613/jair.4016.

[213] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. “A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas”. In: *Inf. Process. Lett.* 8.3 (1979), pp. 121–123. DOI: 10.1016/0020-0190(79)90002-4.

[214] Allen Van Gelder. “Toward Leaner Binary-Clause Reasoning in a Satisfiability Solver”. In: *Annals of Mathematics and Artificial Intelligence* 43.1 (Jan. 2005), pp. 239–253. ISSN: 1573-7470. DOI: 10.1007/s10472-005-0433-5.

[215] Daniel Le Berre. “Exploiting the Real Power of Unit Propagation Lookahead”. In: *Electron. Notes Discret. Math.* 9 (2001), pp. 59–80. DOI: 10.1016/S1571-0653(04)00314-2.

[216] John Franco and Allen Van Gelder. “A Perspective on Certain Polynomial-Time Solvable Classes of Satisfiability”. In: *Discrete Applied Mathematics* 125.2 (Feb. 2003), pp. 177–214. ISSN: 0166-218X. DOI: 10.1016/S0166-218X(01)00358-4.

[217] Ondřej Čepek, Petr Kučera, and Václav Vlček. “Properties of SLUR Formulae”. In: *SOFSEM 2012: Theory and Practice of Computer Science*. Ed. by Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 177–189. ISBN: 978-3-642-27660-6. DOI: 10.1007/978-3-642-27660-6_15.

[218] Matti Järvisalo and Janne H. Korhonen. “Conditional Lower Bounds for Failed Literals and Related Techniques”. In: *Theory and Applications of Satisfiability Testing – SAT 2014*. Ed. by Carsten Sinz and Uwe Egly. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 75–84. ISBN: 978-3-319-09284-3. DOI: 10.1007/978-3-319-09284-3_7.

[219] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. “Which Problems Have Strongly Exponential Complexity?” In: *Journal of Computer and System Sciences* 63.4 (Dec. 2001), pp. 512–530. ISSN: 0022-0000. DOI: 10.1006/jcss.2001.1774.

[220] Adnan Darwiche and Knot Pipatsrisawat. “Complete Algorithms”. In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 101–132. DOI: 10.3233/FAIA200986. URL: https://doi.org/10.3233/FAIA200986.

[221] Patrik Simons, Ilkka Niemelä, and Timo Soininen. “Extending and Implementing the Stable Model Semantics”. In: *Artificial Intelligence* 138.1-2 (2002), pp. 181–234. DOI: 10.1016/S0004-3702(02)00187-X.

[222] Roman Gershman and Ofer Strichman. “Cost-Effective Hyper-Resolution for Preprocessing CNF Formulas”. In: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*. Ed. by Fahiem Bacchus and Toby Walsh. Vol. 3569. Lecture Notes in Computer Science. Springer, 2005, pp. 423–429. DOI: 10.1007/11499107_34.

[223] Per Bjesse and James H. Kukula. “Automatic generalized phase abstraction for formal verification”. In: *ICCAD*. IEEE Computer Society, 2005, pp. 1076–1082.

[224] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. “Efficient implementation of property directed reachability”. In: *FMCAD*. FMCAD Inc., 2011, pp. 125–134.

[225] Michael L. Case, Jason Baumgartner, Hari Mony, and Robert Kanzelman. “Approximate Reachability with Combined Symbolic and Ternary Simulation”. In: *FMCAD*. Ed. by Per Bjesse and Anna Slobodová. FMCAD Inc., 2011, pp. 109–115.

[226] Patrick Cousot. *Méthodes Itératives de Construction et d’approximation de Points Fixes d’opérateurs Monotones Sur Un Treillis, Analyse Sémantique Des Programmes*. 1978.

[227] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “The ASTREÉ Analyzer”. In: *ESOP*. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 21–30.

[228] Carl-Johan H. Seger and Randal E. Bryant. “Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories”. In: *Formal Methods Syst. Des.* 6.2 (1995), pp. 147–189.

[229] Ching-Tsun Chou. “The mathematical foundation of symbolic trajectory evaluation”. In: *CAV’99*. Springer. 1999, pp. 196–207.

[230] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability - Second Edition*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021.

[231] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL*. ACM, 1977, pp. 238–252.

[232] Patrick Cousot and Radhia Cousot. “Basic Concepts of Abstract Interpretation”. In: *IFIP*. Ed. by René Jacquart. Vol. 156. IFIP. Kluwer/Springer, 2004, pp. 359–366. DOI: 10.1007/978-1-4020-8157-6_27.

[233] Agostino Cortesi and Matteo Zanioli. “Widening and Narrowing Operators for Abstract Interpretation”. In: 37.1 (2011), pp. 24–42. DOI: 10.1016/J.CL.2010.09.001.

[234] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[235] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition*. MIT Press, 2018. ISBN: 978-0-262-03883-6.

[236] Hasan Amjad. “Programming a Symbolic Model Checker in a Fully Expansive Theorem Prover”. In: *TPHOLs*. Vol. 2758. Lecture Notes in Computer Science. Springer, 2003, pp. 171–187.

[237] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. “A Fully Verified Executable LTL Model Checker”. In: vol. 2014. 2014.

[238] Christoph Sprenger. “A Verified Model Checker for the Modal μ-calculus in Coq”. In: *TACAS*. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 167–183.

[239] Dirk Beyer, Po-Chun Chien, and Nian-Ze Lee. “Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator”. In: *TACAS (2)*. Vol. 13994. Lecture Notes in Computer Science. Springer, 2023, pp. 152–172.

[240] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. “Correctness witnesses: Exchanging verification results between verifiers”. In: *SIGSOFT FSE*. ACM, 2016, pp. 326–337.

[241] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. “Verification Witnesses”. In: *ACM Trans. Softw. Eng. Methodol.* 31.4 (2022), 57:1–57:69.

[242] Alberto Griggio, Marco Roveri, and Stefano Tonetta. “Certifying Proofs for LTL Model Checking”. In: *FMCAD*. IEEE, 2018, pp. 1–9.

[243] Alberto Griggio, Marco Roveri, and Stefano Tonetta. “Certifying proofs for SAT-based model checking”. In: *Formal Methods Syst. Des.* 57.2 (2021), pp. 178–210.

[244] Tuomas Kuismin and Keijo Heljanko. “Increasing Confidence in Liveness Model Checking Results with Proofs”. In: *Haifa Verification Conference*. Vol. 8244. Lecture Notes in Computer Science. Springer, 2013, pp. 32–43.

[245] Alain Mebsout and Cesare Tinelli. “Proof certificates for SMT-based model checkers for infinite-state systems”. In: *FMCAD*. IEEE, 2016, pp. 117–124.

[246] Kedar S. Namjoshi. “Certifying Model Checkers”. In: *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*. Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Vol. 2102. Lecture Notes in Computer Science. Springer, 2001, pp. 2–13. DOI: 10.1007/3-540-44585-4_2.

[247] Haniel Barbosa, Clark W. Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar. “Generating and Exploiting Automated Reasoning Proof Certificates”. In: *Commun. ACM* 66.10 (2023), pp. 86–95. DOI: 10.1145/3587692.

[248] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark W. Barrett. “Flexible Proof Production in an Industrial-Strength SMT Solver”. In: *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*. Ed. by Jasmin Blanchette, Laura Kovács, and Dirk Pattinson. Vol. 13385. Lecture Notes in Computer Science. Springer, 2022, pp. 15–35. DOI: 10.1007/978-3-031-10769-6_3.

[249] Jochen Hoenicke and Tanja Schindler. “A Simple Proof Format for SMT”. In: *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*. Ed. by David Déharbe and Antti E. J. Hyvärinen. Vol. 3185. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 54–70.

[250] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. “Alethe: Towards a Generic SMT Proof Format (extended abstract)”. In: *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*. Ed. by Chantal Keller and Mathias Fleury. Vol. 336. EPTCS. 2021, pp. 49–54. DOI: 10.4204/EPTCS.336.6.

[251] Marijn Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. “Efficient, Verified Checking of Propositional Proofs”. In: *ITP*. Vol. 10499. Lecture Notes in Computer Science. Springer, 2017, pp. 269–284.

[252] Daniela Kaufmann, Mathias Fleury, Armin Biere, and Manuel Kauers. “Practical algebraic calculus and Nullstellensatz with the checkers Pacheck and Pastèque and Nuss-Checker”. In: *Formal Methods in System Design* (2022), pp. 1–35.

[253] Emily Yu, Armin Biere, and Keijo Heljanko. “Progress in Certifying Hardware Model Checking Results”. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 363–386. DOI: 10.1007/978-3-030-81688-9_17.

[254] Hari Mony, Jason Baumgartner, and Adnan Aziz. “Exploiting Constraints in Transformation-Based Verification”. In: *CHARME*. Vol. 3725. Lecture Notes in Computer Science. Springer, 2005, pp. 269–284.

[255] Jason Baumgartner, Tamir Heyman, Vigyan Singhal, and Adnan Aziz. “Model Checking the IBM Gigahertz Processor: An Abstraction Algorithm for High-Performance Netlists”. In: *CAV*. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 72–83.

[256] Jason Baumgartner, Tamir Heyman, Vigyan Singhal, and Adnan Aziz. “An Abstraction Algorithm for the Verification of Level-Sensitive Latch-Based Netlists”. In: *Formal Methods Syst. Des.* 23.1 (2003), pp. 39–65.

[257] Michael L. Case, Hari Mony, Jason Baumgartner, and Robert Kanzelman. “Enhanced verification by temporal decomposition”. In: *FMCAD*. IEEE, 2009, pp. 17–24.

[258] C. A. J. van Eijk and Jochen A. G. Jess. “Exploiting Functional Dependencies in Finite State Machine Verification”. In: *1996 European Design and Test Conference, ED&TC 1996, Paris, France, March 11-14, 1996*. IEEE Computer Society, 1996, pp. 9–14. DOI: 10.1109/EDTC.1996.494119.

[259] Anatoli Degtyarev and Andrei Voronkov. “Equality Reasoning in Sequent-Based Calculi”. In: *Handbook of Automated Reasoning (in 2 volumes)*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 611–706.

[260] Armin Biere. “Bounded Model Checking”. In: Frontiers in Artificial Intelligence and Applications 336 (2021). Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, pp. 739–764. DOI: 10.3233/FAIA201002.

[261] Masahiro Fujita. “Toward unification of synthesis and verification in topologically constrained logic design”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 2052–2060.

[262] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. “DAG-aware AIG rewriting a fresh look at combinational logic synthesis”. In: *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*. Ed. by Ellen Sentovich. ACM, 2006, pp. 532–535. DOI: 10.1145/1146909.1147048. URL: https://doi.org/10.1145/1146909.1147048.

[263] Alan Mishchenko, Satrajit Chatterjee, Robert K. Brayton, and Niklas Eén. “Improvements to combinational equivalence checking”. In: *ICCAD*. ACM, 2006, pp. 836–843.

[264] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K Brayton. *FRAIGs: A unifying representation for logic synthesis and verification*. Tech. rep. ERL Technical Report, 2005.

[265] Qi Zhu, Nathan Kitchen, Andreas Kuehlmann, and Alberto L. Sangiovanni-Vincentelli. “SAT sweeping with local observability don’t-cares”. In: *DAC*. ACM, 2006, pp. 229–234.

[266] Armin Biere and Mathias Fleury. “Mining Definitions in Kissat with Kittens”. In: *Workshop on the Pragmatics of SAT 2021*. 2021. URL: http://www.pragmaticsofsat.org/2021/.

[267] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. “Definability for model counting”. In: *Artif. Intell.* 281 (2020), p. 103229. DOI: 10.1016/j.artint.2019.103229.

[268] Alessandro Padoa. “Essai d’une théorie algébrique des nombres entiers, précédé d’une Introduction logique à une theorie déductive quelconque”. In: *Bibliothèque du Congrès international de philosophie*. Vol. 3. 1901, pp. 309–365.

[269] Friedrich Slivovsky. “Interpolation-Based Semantic Gate Extraction and Its Applications to QBF Preprocessing”. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 508–528. DOI: 10.1007/978-3-030-53288-8_24.

[270] Armin Biere and Robert Brummayer. “Consistency Checking of All Different Constraints over Bit-Vectors within a SAT Solver”. In: *FMCAD*. IEEE, 2008, pp. 1–4.

[271] Certifaiger. “Certifaiger”. In: (2021). http://fmv.jku.at/certifaiger.

[272] Armin Biere and Koen Claessen. “Hardware Model Checking Competition 2010”. In: (2010).
http://fmv.jku.at/hwmcc10/.

[273] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[274] Larry Paulson and Tobias Nipkow. *The Sledgehammer: Let automatic theorem provers write your Isabelle scripts*. 2023.

[275] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. “Witness validation and stepwise testification across software verifiers”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 721–733.

[276] Matt Kaufman, Andrew Martin, and Carl Pixley. “Design constraints in symbolic model checking”. In: *Computer Aided Verification: 10th International Conference, CAV’98 Vancouver, BC, Canada, June 28–July 2, 1998 Proceedings 10*. Springer. 1998, pp. 477–487.

[277] Yoav Hollander, Matthew Morley, and Amos Noy. “The e language: A fresh separation of concerns”. In: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 38*. IEEE. 2001, pp. 41–50.

[278] Rajeev Alur and Thomas A Henzinger. “Reactive modules”. In: *Formal methods in system design* 15 (1999), pp. 7–48.

[279] Sharad Malik. “Analysis of cyclic combinational circuits”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 13.7 (1994), pp. 950–956.

[280] Jie-Hong Roland Jiang, Alan Mishchenko, and Robert K. Brayton. “On breakable cyclic definitions”. In: *ICCAD*. IEEE Computer Society / ACM, 2004, pp. 411–418.

[281] Marc D Riedel. *Cyclic combinational circuits*. California Institute of Technology, 2004.

[282] Gianpiero Cabodi, Paolo Camurati, Luz Amanda Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. “Speeding up model checking by exploiting explicit and hidden verification constraints”. In: *DATE*. IEEE, 2009, pp. 1686–1691.

[283] Koen Claessen and Niklas Sörensson. “A Liveness Checking Algorithm That Counts”. In: *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*. Ed. by Gianpiero Cabodi and Satnam Singh. IEEE, 2012, pp. 52–59.

[284] MS Jahanpour and OA Mohamed. “Automatic generation of model checking properties and constraints from production based specification”. In: *The 2004 47th Midwest Symposium on Circuits and Systems, 2004. MWSCAS’04.* Vol. 3. IEEE. 2004, pp. iii–435.

[285] Jun Yuan, Ken Albin, Adnan Aziz, and Carl Pixley. “Constraint synthesis for environment modeling in functional verification”. In: *Proceedings of the 40th annual Design Automation Conference*. 2003, pp. 296–299.

[286] Eugene Goldberg, Matthias Güdemann, Daniel Kroening, and Rajdeep Mukherjee. “Efficient Verification of Multi-Property Designs (The Benefit of Wrong Assumptions)”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. Ed. by Jan Madsen and Ayse K. Coskun. IEEE, 2018, pp. 43–48. DOI: 10.23919/DATE.2018.8341977.

[287] Sourav Das, Aritra Hazra, Pallab Dasgupta, Sudipta Kundu, and Himanshu Jain. “PURSE: Property Ordering Using Runtime Statistics for Efficient Multi - Property Verification”. In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2024, pp. 1–6. DOI: 10.23919/DATE58400.2024.10546895.

[288] Zhengqi Yu, Armin Biere, and Keijo Heljanko. “Certifying Hardware Model Checking Results”. In: *ICFEM*. Vol. 11852. Lecture Notes in Computer Science. Springer, 2019, pp. 498–502.

[289] Arie Gurfinkel and Alexander Ivrii. “K-induction without unrolling”. In: *FM-CAD*. IEEE, 2017, pp. 148–155.

[290] Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. “Horn Clause Solvers for Program Verification”. In: *Fields of Logic and Computation II*. Vol. 9300. Lecture Notes in Computer Science. Springer, 2015, pp. 24–51.

[291] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. “The Kind 2 Model Checker”. In: *CAV (2)*. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 510–517.

[292] Leander Tentrup. “CAQE and quabs: Abstraction based QBF solvers”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 11.1 (2019), pp. 155–210.

[293] Charles Jordan, Will Klieber, and Martina Seidl. “Non-CNF QBF Solving with QCIR”. In: *AAAI Workshop: Beyond NP*. Vol. WS-16-05. AAAI Workshops. AAAI Press, 2016.

[294] Narendra Shenoy and Richard Rudell. “Efficient implementation of retiming”. In: *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design* (2003), pp. 615–630.

[295] Gianpiero Cabodi, Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Danilo Vendraminetto, Armin Biere, and Keijo Heljanko. “Hardware Model Checking Competition 2014: An Analysis and Comparison of Solvers and Benchmarks”. In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 135–172. DOI: 10.3233/SAT190106. URL: https://doi.org/10.3233/sat190106.

[296] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. “Btor2 , BtorMC and Boolector 3.0”. In: *CAV (1)*. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 587–595.

[297] D. Beyer. “State of the Art in Software Verification and Witness Validation: SV-COMP 2024”. In: *Proc. TACAS (3)*. LNCS 14572. Springer, 2024, pp. 299–329. DOI: 10.1007/978-3-031-57256-2_15.

[298] Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins. “MaxSAT evaluation 2020: Solver and benchmark descriptions”. In: (2020).

[299] Luca Pulina and Martina Seidl. “The 2016 and 2017 QBF solvers evaluations (QBFEVAL’16 and QBFEVAL’17)”. In: *Artif. Intell.* 274 (2019), pp. 224–248.

[300] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. “The SMT competition 2015–2018”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 11.1 (2019), pp. 221–259.

[301] Geoff Sutcliffe. “Proceedings of the 12th IJCAR ATP System Competition (CASC-J12)”. In: (2024).

[302] Salomé Eriksson, Gabriele Röger, and Malte Helmert. “Unsolvability Certificates for Classical Planning”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*. Ed. by Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith. AAAI Press, 2017, pp. 88–97.

[303] Salomé Eriksson and Malte Helmert. “Certified Unsolvability for SAT Planning with Property Directed Reachability”. In: (2020). Ed. by J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi, pp. 90–100.

[304] Sylvain Conchon, Alain Mebsout, and Fatiha Zaïdi. “Certificates for Parameterized Model Checking”. In: *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. Ed. by Nikolaj Bjørner and Frank S. de Boer. Vol. 9109. Lecture Notes in Computer Science. Springer, 2015, pp. 126–142.

[305] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

[306] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. “The Lean theorem prover (system description)”. In: *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer. 2015, pp. 378–388.

[307] Per Bjesse. “Word-Level Sequential Memory Abstraction for Model Checking”. In: *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*. Ed. by Alessandro Cimatti and Robert B. Jones. IEEE, 2008, pp. 1–9. DOI: 10.1109/FMCAD.2008.ECP.20.

[308] Henry A. Kautz and Bart Selman. “Planning as Satisfiability”. In: *ECAI*. 1992, pp. 359–363.