Strong Induction in Hardware Model Checking

by

Hari Govind V K

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical And Computer Engineering

Waterloo, Ontario, Canada, 2019

© Hari Govind V K 2019

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some of the text, figures and tables in this thesis are restated from our CAV 2019 paper [49]. Some of the source code for KAVY was written by Arie Gurfinkel.

Abstract

Symbolic model checking is a widely used technique for automated verification of both hardware and software systems. Unbounded SAT-based Symbolic Model Checking (SMC) algorithms are very popular in hardware verification. The principle of strong induction is one of the first techniques for SMC. While elegant and simple to apply, properties as such can rarely be proven using strong induction and when they can be strengthened, there is no effective strategy to guess the depth of induction. It has been mostly displaced by techniques that compute inductive strengthenings based on interpolation and property directed reachability (PDR). In this thesis, we prove that strong induction is more concise than induction. We then present kAVY, an SMC algorithm that effectively uses strong induction to guide interpolation and PDR-style incremental inductive invariant construction. Unlike pure strong induction, kAVY uses PDR-style generalization to compute and strengthen an inductive trace. Unlike pure PDR, kAVY uses relative strong induction to construct an inductive invariant. The depth of induction is adjusted dynamically by minimizing a proof of unsatisfiability. We have implemented kAVY within the AVY Model Checker and evaluated it on HWMCC instances. Our results show that kAVY is more effective than both AVY and PDR, and that using strong induction leads to faster running time and solving more instances. Further, on a class of benchmarks, called *shift*, kAVY is orders of magnitude faster than AVY, PDR and pure strong induction.

Acknowledgements

I would like to thank Professor Arie Gurfinkel and Professor Vijay Ganesh for sharing their expertise in the subject, availability and giving valuable advices for this research. It goes without saying that this document would not be possible without their help and guidence.

I would like to thank Professor Nancy Day and Professor Richard Trefler for attending the seminar, reviewing the thesis and providing valuable feedback on both occasions.

I would like to express my gratitude to my Mother, Father, Sister and Brother for their constant support throughout my life, especially during the Master's.

Special thanks to Jakub Kuderski and Chunxiao (Ian) Li for their company and frequent discussions about this research topic.

Thank God.

Table of Contents

List of Tables viii

List of Figures ix

Nomenclature x

1 Introduction **1**

1.1 Contributions 4

2 Background **5**

2.1 SAT-based model checking 7

2.1.1 Model checking using strong induction 8

2.1.2 Model checking using Interpolation 8

2.1.3 Incremental construction of inductive invariants 9

2.1.4 PDR 12

2.1.5 Avy 13

3 Separation between strong induction and induction **16**

3.1 Separation between 1-induction and 2-induction 17

3.1.1 Validating certificates for strong induction and induction 19

3.1.2 PDR vs strong induction on C_{xor_n} 20

3.2 Generalization of the counter circuit 20

4 Kavy **22**
4.1 Extending a trace with strong induction 24
4.2 Searching for the maximal SEL 29
4.3 Evalution 31

5 Related work, Conclusions and Future work **35**

References **37**

List of Tables

4.1 Summary of solved instances 32

List of Figures

1.1 An example system 3

3.1 States of the counter circuit 18

4.1 Runtime comparison on SAFE instances 31

4.2 Comparing kAvy against Avy, Pdr and vanilla 33

4.3 Comparing top-down and bottom-up 34

Nomenclature

Bad	The bad state in a transision system
$\mathcal{K}(\boldsymbol{F})$	The set of all strong externsion levels of $\boldsymbol{F}$
$\mathcal{W}(\boldsymbol{F})$	The set of all extension levels of $\boldsymbol{F}$
$Init$	The initial state in a transistion system
$Tr[\![\boldsymbol{F}^i]\!]^k$	Characteristic formula for SEL (i, k)
$\textsc{seqItp}(\boldsymbol{A})$	A sequence interpolant of $\boldsymbol{A}$
Tr	The transistion relation in a transistion system
$Tr[\varphi]_M^N$	An M-to-N-unrolling of a transision system, where φ holds in all intermediate states
$Tr[\boldsymbol{F}]$	An unrolling of Tr such that F_i holds at step i
Tr_M^N	An M-to-N-unrolling of a transision system. M is skipped when $M = 0$
$\boldsymbol{F}$	A sequence of formulas $[F_0, F_1, \ldots, F_N]$
$\boldsymbol{F}^k$	k-suffix of $\boldsymbol{F}$. $\boldsymbol{F}^k = [F_k, \ldots, F_N]$
SMC	SAT-based Model Checking

Chapter 1

Introduction

This thesis concerns with the problem of SAFETY: given a state transition system and a set of bad states, check whether there exists a finite length execution path that leads to a bad state. Functional verification of hardware circuits can be achieved by modeling hardware circuits as state transition systems, marking undesired behaviors as bad states and solving the problem of SAFETY. A similar approach can be used to check equivalence between two designs, verify software programs and many other important problems [55, 27]. The problem has come up when designing components in nuclear power plants [66], preventing crashes in autonomous systems [72, 37] and spacecraft [7], scheduling trains [58] and many other areas.

There are several approaches to solve SAFETY. Binary Decision Diagrams, abstract interpretation, explicit state model checking, interactive theorem proving and symbolic model checking are some [29]. Symbolic model checking techniques express the transition system and the set of bad states in logic and exhaustively check for a counterexample or a proof of SAFETY. While the problem is undecidable in general, the propositional variant is PSPACE-complete [36]. Owing to algorithmic advances and the advent of SAT solvers, SAT based (unbounded) model checking algorithms (SMC) have received particular attention in recent years [55].

SAFETY can be established using the principle of induction. The certificate for induction is a SAFE inductive invariant. An inductive invariant is a formula that is (1) satisfied by all initial states in the transition system (base case) (2) closed under a step of transition (inductive case). It is SAFE when it implies the negation of bad states (called property). The problem of proving SAFETY can be reduced to finding the right inductive strengthening of the property.

SAFETY can also be established using the principle of strong induction[1]. The certificate for strong induction is a SAFE k-inductive invariant: a generalization of an inductive invariant that extends the base- and inductive-cases to k steps of a transition system [71]. Induction is contained within strong induction. When restricted to loop-free paths, the property itself is a certificate for strong induction. No such claims can be made for induction.

In SMC, strong induction and induction have the same deductive power: a certificate P for strong induction can be converted into a certificate Q for induction [41]. However, in the worst case Q may be exponentially larger than P [17]. The first contribution of this thesis is to construct a family of transition systems in which this behavior is observed. In Chapter 3, we show that there are transition systems whose smallest 1-inductive certificate is exponentially larger than the smallest strong inductive certificate.

Unlike other SMC techniques, strong induction reduces model checking to pure SAT that does not require any additional features such as solving with assumptions [34], interpolation [61], resolution proofs [43], Maximal Unsatisfiable Subsets (MUS) [9], etc. It easily integrates with existing SAT-solvers and immediately benefits from any improvements in heuristics [57, 56], pre- and in-processing [44], and parallel solving [6]. The simplicity of applying strong induction made it the go-to technique for SMT-based infinite-state model checking [25, 33, 46]. In that context, it is particularly effective in combination with invariant synthesis [50, 38]. Moreover, for some theories, strong induction is strictly stronger than 1-induction [46]: there are properties that are k-inductive, but have no 1-inductive strengthening.

Despite these advantages, strong induction has mostly been displaced by techniques based on (1)-induction. The exponential growth in size of invariants is rarely observed in practice [63]. Furthermore, the SAT queries get very hard as k increases and usually succeed only for rather small values of k.

Property Directed Reachability (PDR) [20, 35] is a very successful technique that *incrementally* constructs an inductive strengthening of the property. PDR constructs the proof by repeatedly generating predecessors of bad states and blocking them using inductive generalization. This technique of incremental invariant construction has inspired many similar, effective, algorithms in both software [46, 28, 52] and hardware [77, 10, 40] model checking. AVY [77] is a successful technique which uses interpolation [31] to guide PDR-style incremental inductive invariant construction.

[1]The principle of strong induction has a particular definition in mathematics. Our usage does not conform with this definition. However, we still use the term strong induction to emphasize that the value of k in k-induction is always dynamic

```
reg [7:0] c = 0;
always
  if(c == 64)
    c <= 0;
  else
    c <= c + 1;
end
assert property (c < 66);
```

Figure 1.1: An example system in verilog.

A recent work [41] shows that strong induction can be integrated in PDR. However, [41] argues that strong induction is hard to control in the context of PDR since choosing a proper value of k is difficult. A wrong choice leads to a form of state enumeration. In [41], k is fixed to 5, and regular induction is used as soon as 5-induction fails.

In Chapter 4, we present KAVY, a novel SMC algorithm that effectively uses strong induction to guide PDR-style inductive invariant construction. As many state-of-the-art SMC algorithms, KAVY iteratively constructs candidate inductive invariants for a given safety property P. However, the construction of these candidates is driven by strong induction. Whenever P is known to hold up to a bound N, KAVY searches for the smallest $k \leq N + 1$, such that either P or some of its strengthening is k-inductive. Once it finds the right k and strengthening, it computes a 1-inductive strengthening.

It is convenient to think of modern SMC algorithms (e.g., PDR and AVY), and strong induction, as two ends of a spectrum. On the one end, modern SMC algorithms fix k to 1 and *search* for a 1-inductive strengthening of P. While on the opposite end, strong induction fixes the strengthening of P to be P itself and *searches* for a k such that P is k-inductive. KAVY *dynamically* explores this spectrum, exploiting the interplay between finding the right k and finding the right strengthening.

As an example, consider the system in Fig. 1.1 that counts up to 64 and resets. The property, $p : c < 66$, is 2-inductive. Both PDR and AVY iteratively guess a 1-inductive strengthening of p. In the worst case, they require at least 64 iterations. On the other hand, KAVY determines that p is 2-inductive after 2 iterations, *computes* a 1-inductive invariant $(c \neq 65) \wedge (c < 66)$, and terminates.

KAVY builds upon the foundations of AVY [77]. AVY first uses Bounded Model Checking [13] (BMC) to prove that the property P holds up to bound N. Then, it uses a sequence interpolant [76] and PDR-style inductive-generalization [20] to construct 1-inductive strengthening candidate for P. We emphasize that using strong induction to construct 1-inductive candidates allows KAVY to efficiently utilize many principles from PDR and AVY. While maintaining k-inductive candidates might seem attractive (since they may be

smaller), they are also much harder to generalize effectively [20].

We implemented kAvy in the Avy Model Checker, and evaluated it on the benchmarks from the Hardware Model Checking Competition (HWMCC). Our experiments show that kAvy significantly improves the performance of Avy and solves more examples than both Pdr and Avy. For a specific family of examples from [53], kAvy exhibits nearly constant time performance, compared to an exponential growth of Avy, Pdr, and strong induction (see Figure 4.1b in Section 4.3). This further emphasizes the effectiveness of efficiently integrating strong induction into modern SMC.

1.1 Contributions

This thesis makes the following contributions

- We provide a constructive proof of an exponential separation between the certificates for strong induction and induction (Chapter 3)
- We propose an algorithm which uses strong induction to effectively guide interpolation and Pdr-style inductive invariant generation and prove its correctness (Chapter 4)
- We implement in the algorithm and evaluate it on a wide benchmark suite (Section 4.3)

Chapter 2

Background

In this chapter, we present notation and background that is required for the rest of the thesis.

safety of finite state transition systems. A symbolic finite state transition system T is a tuple $(\bar{v}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$, where $\bar{v}$ is a set of Boolean *state* variables. A state of the system is a complete valuation to all variables in $\bar{v}$ (i.e., the set of states is $\{0,1\}^{|\bar{v}|}$). We write $\bar{v}' = \{v' \mid v \in \bar{v}\}$ for the set of *primed* variables, used to represent the next state. Init and Bad are formulas over $\bar{v}$ denoting the set of initial states and bad states, respectively, and Tr is a formula over $\bar{v} \cup \bar{v}'$, denoting the transition relation. With abuse of notation, we use formulas and the sets of states (or transitions) that they represent interchangeably. In addition, we sometimes use a state s to denote the formula (cube) that characterizes it. For a formula φ over $\bar{v}$, we use $\varphi(\bar{v}')$, or φ' in short, to denote the formula in which every occurrence of $v \in \bar{v}$ is replaced by $v' \in \bar{v}'$. For simplicity of presentation, we assume that the property $P = \neg \mathit{Bad}$ is true in the initial state, that is $\mathit{Init} \Rightarrow P$. We ignore $\bar{v}$ from the tuple $(\bar{v}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$ for T if it is unimportant.

Given a formula $\varphi(\bar{v})$, an M-to-N-*unrolling* of T, where φ holds in all intermediate states is defined by the formula:

$$\mathit{Tr}[\varphi]_M^N = \bigwedge_{i=M}^{N-1} \varphi(\bar{v}_i) \wedge \mathit{Tr}(\bar{v}_i, \bar{v}_{i+1}) \tag{2.1}$$

We write $\mathit{Tr}[\varphi]^N$ when $M = 0$ and Tr_M^N when $\varphi = \top$.

A transition system T is UNSAFE iff there exists a state $s \in Bad$ s.t. s is reachable, and is SAFE otherwise. Equivalently, T is UNSAFE iff there exists a number N such that the following *unrolling* formula is satisfiable:

$$Init(\bar{v}_0) \wedge Tr^N \wedge Bad(\bar{v}_N) \tag{2.2}$$

T is SAFE if no such N exists. Whenever T is UNSAFE and $s_N \in Bad$ is a reachable state, the path from $s_0 \in Init$ to s_N is called a *counterexample*.

Induction. SAFETY can be established using the principle of induction. A certificate for SAFETY is a SAFE *inductive invariant*. An *inductive invariant* is a formula *Inv* that satisfies initiation and is inductive:

$$Init(\bar{v}) \Rightarrow Inv(\bar{v}) \tag{2.3}$$

$$Inv(\bar{v}) \wedge Tr(\bar{v}, \bar{v}') \Rightarrow Inv(\bar{v}') \tag{2.4}$$

An inductive invariant $Inv(\bar{v})$ is SAFE if it satisfies :

$$Inv(\bar{v}) \Rightarrow P(\bar{v}) \tag{2.5}$$

We say that a formula φ is *inductive relative* to a formula F if it satisfies initiation and $Tr[\varphi \wedge F] \Rightarrow \varphi(\bar{v}_1)$.

The set of reachable states is an inductive invariant in all transition systems. Therefore, a transition system is SAFE iff it admits a SAFE inductive invariant.

Strong Induction. According to the principle of strong induction, SAFETY can be established by providing a SAFE *k-inductive invariant* as a certificate. *k-induction* is a generalization of the notion of an inductive invariant. A formula *Inv* is *k-invariant* in a transition system T if it is true in the first k steps of T. That is, the following formula is valid:

$$Init(\bar{v}_0) \wedge Tr^k \Rightarrow \left(\bigwedge_{i=0}^{k} Inv(\bar{v}_i) \right) \tag{2.6}$$

A formula *Inv* is a *k-inductive invariant* iff *Inv* is a $(k-1)$-invariant and is inductive after k steps of T, i.e., the following formula is valid: $Tr[Inv]^k \Rightarrow Inv(\bar{v}_k)$. We say that a formula φ is *k-inductive relative* to F if it is a $(k-1)$-invariant and $Tr[\varphi \wedge F]^k \Rightarrow \varphi(\bar{v}_k)$.

Compared to simple induction, k-induction strengthens the hypothesis in the induction step: *Inv* is assumed to hold between steps 0 to $k-1$ and is established in step k. Whenever $Inv \Rightarrow P$, we say that *Inv* is a SAFE k-inductive invariant. An inductive invariant is a 1-inductive invariant.

Theorem 1. *Given a transition system T. There exists a* SAFE *1-inductive invariant w.r.t. T iff there exists a* SAFE *k-inductive invariant w.r.t. T.*

Theorem 1 states that the strong induction principle is as complete as the induction principle. One direction of the proof is trivial (since we can take $k = 1$). The other direction has been proven in [41]. This can be strengthened further: for every k-inductive invariant Inv_k there exists a 1-inductive strengthening Inv_1 such that $Inv_1 \Rightarrow Inv_k$. Theoretically Inv_1 might be exponentially bigger than Inv_k. In practice, both invariants tend to be of similar size.

The SAFETY verification problem is to decide whether a transition system T is SAFE or UNSAFE, i.e., whether there exists a SAFE k-inductive invariant or a counterexample.

Craig Interpolation [31]. Given a pair of inconsistent formulas (A, B) (i.e., $A \wedge B \models \bot$), a *Craig interpolant* [31] for (A, B) is a formula I such that: (a) $A \Rightarrow I$, (b) $I \Rightarrow \neg B$, and (c) I is over variables shared between A and B. Intuitively, interpolants are over-approximations of A which contradict B. We will see that they are useful in computing over-approximations of reachable states.

We use an extension of Craig Interpolants to sequences, which is common in Model Checking. Let $\boldsymbol{A} = [A_1, \ldots, A_N]$ be a sequence of formulas such that $A_1 \wedge \cdots \wedge A_N$ is unsatisfiable. A *sequence interpolant* $\boldsymbol{I} = \textsc{seqItp}(\boldsymbol{A})$ for $\boldsymbol{A}$ is a sequence of formulas $\boldsymbol{I} = [I_2, \ldots, I_N]$ such that (a) $A_1 \Rightarrow I_2$, (b) $\forall 1 < i < N \cdot I_i \wedge A_i \Rightarrow I_{i+1}$, (c) $I_N \wedge A_N \Rightarrow \bot$, and (d) I_i is over variables that are shared between the corresponding prefix and suffix of $\boldsymbol{A}$.

2.1 SAT-based model checking

In this section, we give a brief overview of SAT-based Model Checking algorithms: strong induction [71], interpolation [61], PDR [20, 35], and AVY [77]. We fix a symbolic transition system $T = (\bar{v}, Init, Tr, Bad)$.

2.1.1 Model checking using strong induction

Algorithm 1 depicts model checking using the principle of strong induction. The algorithm iteratively checks base-case (line 3) and the inductive-case (line 4) for increasing values of k. Each successful check of the base-case establishes the absence of counterexamples up to depth k. If the base-case fails, the satisfying assignment is a valid counterexample for the system. Spurious counterexamples to induction can occur if there are loops that lead to *Bad*, but are unreachable from the initial states. Thus, for the inductive-case, the algorithm adds unique path constraints for completeness: $unique(\bar{v}, k) \equiv \forall i, j \cdot (0 \leq i, j \leq k \wedge i \neq j) \Rightarrow \neg \bigwedge_{v \in \bar{v}} v_i = v_j$. A simple encoding of this in CNF requires k^2 clauses. Once both checks succeed for some value of k, the algorithm returns SAFE, indicating that the property is k-inductive.

Algorithm 1: Model checking using strong induction

Input: A transition system $T = (Init, Tr, Bad)$
Output: SAFE/UNSAFE

```
1 k ← 1
2 repeat
3     if ¬isSat (Init(v̄_0) ∧ Tr[¬Bad]^k ∧ Bad(v̄_k)) then
4         if ¬isSat (Tr[¬Bad]^k ∧ unique(v̄, k) ∧ Bad(v̄_k)) then
5             return SAFE
6     else
7         return UNSAFE
8     k ← k + 1
9 until ∞
```

2.1.2 Model checking using Interpolation

The SAT queries in Algorithm 1 grow quadratically with k. This restricts the algorithm to be effective only for small values of k [41]. Alternately, SAFETY can be proven by computing a 1-inductive strengthening of the property. Algorithm 2 uses interpolation to construct such an inductive strengthening. The algorithm maintains an over-approximation of reachable states and at each iteration, checks whether *Bad* is reachable from the over-approximation (line 4).

If *Bad* is reachable, there is an actual counterexample of k-steps if the over-approximation is equivalent to *Init* (line 9). If the over-approximation is not equivalent to *Init*, it admits a spurious counterexample and the process is restarted at a larger depth (line 10).

If *Bad* is unreachable, the algorithm refines the computed set of over-approximations by computing an interpolant (line 5). A fixed point is reached when the computed set of over-approximations is closed under the transition relation. The algorithm terminates when this happens (line 6).

Algorithm 2: Model checking using interpolation. $ITP(A, B)$ denotes the interpolant of $A \wedge B$.

```
   Input: A transition system T = (Init, Tr, Bad)
   Output: SAFE/UNSAFE
1  R ← Init
2  k ← 1
3  repeat
4     if ¬isSat(R(v̄_0) ∧ Tr[P]^k ∧ Bad(v̄_k)) then
         // Compute interpolant
5        F ← ITP(R(v̄_0) ∧ Tr_0^1, Tr_1^{k+1} ∧ Bad(v̄_{k+1}))
6        if F ⇒ R then return SAFE
7        else
8           R ← R ∨ F(v̄_1 \ v̄_0)
9     else if R ≡ Init then return UNSAFE
10    else
11       R ← Init
12       k ← k + 1
13 until ∞
```

2.1.3 Incremental construction of inductive invariants

Algorithm 2 relies on interpolation to generate the inductive invariant. However, interpolation algorithms are not guided by any search for inductive invariants. Thus, the interpolant generated by subsequent calls to the algorithm could be significantly different, delaying convergence. The underlying heuristics could favor smaller resolution proofs in favor of smaller inductive invariants. Furthermore, the algorithm resets whenever the computed candidate inductive invariant admits a spurious counterexample, leading to wasted

effort until the correct value of k is reached. IC3 [20][1] was the first algorithm to propose an incremental construction of inductive invariants by guiding the underlying SAT solver to prune predecessors of *Bad*. The algorithm won third place in the Hardware Model Checking Competition (HWMCC) in 2011, and inspired Avy, kAvy and many other algorithms in both software and hardware model checking.

The main data-structure of Pdr and Avy is a sequence of candidate invariants, called an *inductive trace*. An *inductive trace*, or simply a trace, is a sequence of formulas $\boldsymbol{F} = [F_0, \ldots, F_N]$ that satisfy the following two properties:

$$Init(\bar{v}) = F_0(\bar{v}) \qquad\qquad \forall 0 \leq i < N \cdot F_i(\bar{v}) \wedge Tr(\bar{v}, \bar{v}') \Rightarrow F_{i+1}(\bar{v}') \tag{2.7}$$

An element F_i of a trace is called a *frame*. The index of a frame is called a *level*. $\boldsymbol{F}$ is *clausal* when all its elements are in CNF. For convenience, we view a frame as a set of clauses, and assume that a trace is padded with $\top$ until the required length. The *size* of $\boldsymbol{F} = [F_0, \ldots, F_N]$ is $|\boldsymbol{F}| = N$. For $k \leq N$, we write $\boldsymbol{F}^k = [F_k, \ldots, F_N]$ for the k-suffix of $\boldsymbol{F}$.

A trace $\boldsymbol{F}$ of size N is *stronger* than a trace $\boldsymbol{G}$ of size M iff $\forall 0 \leq i \leq \min(N, M) \cdot F_i(\bar{v}) \Rightarrow G_i(\bar{v})$. A trace is SAFE if each F_i is SAFE: $\forall i \cdot F_i \Rightarrow \neg Bad$; *monotone* if $\forall 0 \leq i < N \cdot F_i \Rightarrow F_{i+1}$. In a monotone trace, a frame F_i over-approximates the set of states reachable in up to i steps of the Tr. A trace is closed if $\exists 1 \leq i \leq N \cdot F_i \Rightarrow \left(\bigvee_{j=0}^{i-1} F_j\right)$. A *predecessor sequence* is a sequence of states $s_1, s_2, \ldots, s_k$ such that $\forall 1 \leq i < k \cdot (s_i, s_{i+1}) \in Tr$ and $s_k \in Bad$. A SAFE trace of size N, blocks all N length predecessor sequences. That is, there exists no predecessor sequence $s_0, s_2, \ldots, s_N$ such that $s_N \in F_N$.

We define an unrolling formula of a trace $\boldsymbol{F} = [F_0, \ldots, F_N]$ as follows:

$$Tr[\boldsymbol{F}] = \bigwedge_{i=0}^{|F|} F_i(\bar{v}_i) \wedge Tr(\bar{v}_i, \bar{v}_{i+1}) \tag{2.8}$$

We write $Tr[\boldsymbol{F}^k]$ to denote an unrolling of a k-suffix of $\boldsymbol{F}$:

$$Tr[\boldsymbol{F}^k] = \bigwedge_{i=k}^{|F|} F_i(\bar{v}_i) \wedge Tr(\bar{v}_i, \bar{v}_{i+1}) \tag{2.9}$$

Intuitively, $Tr[\boldsymbol{F}^k]$ is satisfiable iff there is a k-step execution of the Tr that is consistent with the k-suffix $\boldsymbol{F}^k$. If a transition system T admits a SAFE trace $\boldsymbol{F}$ of size $|\boldsymbol{F}| = N$, then T does not have counterexamples of length less than or equal to N.

[1] It was suggested to be renamed to Property Directed Reachability (Pdr) in [35]

Definition 1. *A* SAFE *trace* $\boldsymbol{F}$*, with* $|\boldsymbol{F}| = N$ *is extendable with respect to level* $0 \leq i \leq N$ *iff there exists a* SAFE *trace* $\boldsymbol{G}$ *stronger than* $\boldsymbol{F}$ *such that* $|\boldsymbol{G}| > N$ *and* $F_i \wedge Tr \Rightarrow G_{i+1}$.

$\boldsymbol{G}$ and the corresponding level i are called an *extension trace* and an *extension level* of $\boldsymbol{F}$, respectively. Note that all the frames after extension level are stronger in extension trace: $\forall k > i \cdot G_k \Rightarrow F_k$. Both PDR and AVY work by iteratively extending a given SAFE trace $\boldsymbol{F}$ of size N to a SAFE trace of size $N+1$.

An extension trace is not unique, but there is a largest extension level. We denote the set of all extension levels of $\boldsymbol{F}$ by $\mathcal{W}(\boldsymbol{F})$. Note that the existence of an extension level i implies that an unrolling of the i-suffix does not contain any *Bad* states:

Lemma 1. *Let* $\boldsymbol{F}$ *be a* SAFE *trace. Then,* i*,* $0 \leq i \leq N$*, is an extension level of* $\boldsymbol{F}$ *iff the formula* $Tr[\boldsymbol{F}^i] \wedge Bad(\bar{v}_{N+1})$ *is unsatisfiable.*

Proof. AVY (Section 2.1.5) proves the right-to-left direction by providing a method to construct an extension trace from the unsatisfiability of $Tr[\boldsymbol{F}^i] \wedge Bad(\bar{v}_{N+1})$. For the left-to-right direction, we give a proof by contradiction. Let $\boldsymbol{G}$ be an extension trace at extension level i of $\boldsymbol{F}$. Since $\boldsymbol{G}$ is SAFE up to $(N+1)$, G_{i+1} is strong enough to block all *predecessor sequences* of length $(N+1-i)$. We show that if $Tr[\boldsymbol{F}^i] \wedge Bad(\bar{v}_{N+1})$ is satisfiable, G_{i+1} admits such a predecessor sequence, there by arriving at a contradiction.

If $Tr[\boldsymbol{F}^i] \wedge Bad(\bar{v}_{N+1})$ is satisfiable, so is the weaker formula $F_i \wedge Tr_i^{N+1} \wedge Bad(\bar{v}_{N+1})$. Let $s_i, s_{i+1}, \ldots s_{N+1}$ be the $(N+2-i)$ length predecessor sequence that satisfies the weaker formula. s_i satisfies F_i. Since $F_i \wedge Tr \Rightarrow G_{i+1}$, s_{i+1} necessarily satisfies G_{i+1}. That is $s_{i+1}, s_{i+2}, \ldots s_{N+1}$ is a $(N+1-i)$ length predecessor sequence not blocked by G_{i+1}. This contradicts the claim that G_{i+1} is strong enough to block any such predecessor sequence. Hence $F_i \wedge Tr_i^{N+1} \wedge Bad(\bar{v}_{N+1})$ is unsatisfiable. Therefore the stronger formula $Tr[\boldsymbol{F}^i] \wedge Bad(\bar{v}_{N+1})$ is also unsatisfiable.

□

Notice that the unsatisfiability is caused due to F_i. Not all the frames in $\boldsymbol{F}^i$ are necessary to prove the lemma. However, as we will see, the suffix will make it easier to construct the extension trace.

Example 1. *For Fig. 1.1,* $\boldsymbol{F} = [c = 0, c < 66]$ *is a* SAFE *trace of size* 1*. The formula* $Tr[\boldsymbol{F}^1] \wedge Bad$*:* $((c < 66) \wedge Tr \wedge (c' \geq 66))$ *is satisfiable. Therefore, there does not exist an extension trace at level* 1*. Since* $Tr[\boldsymbol{F}^0] \wedge Bad$*:* $((c = 0) \wedge Tr \wedge (c' < 66) \wedge Tr' \wedge (c'' \geq 66))$ *is unsatisfiable, the trace is extendable at level* 0*. For example, a valid extension trace at level* 0 *is* $\boldsymbol{G} = [c = 0, c < 2, c < 66]$*.*

Both PDR and AVY iteratively extend a SAFE trace either until the extension is closed or a counterexample is found. However, they differ in how exactly the trace is extended. In the rest of this section, we present PDR and AVY through the lens of extension level.

2.1.4 PDR

PDR maintains a monotone, clausal trace $\boldsymbol{F}$ with *Init* as the first frame (F_0). The trace $\boldsymbol{F}$ is extended by recursively computing and blocking (if possible) states that are part of predecessor sequences (called bad states). A bad state is blocked at the largest level possible. Algorithm 3 shows PDRBLOCK, the backward search procedure that identifies and blocks predecessor sequences. PDRBLOCK maintains a queue of states and the levels at which they have to be blocked. The smallest level at which blocking occurs is tracked in order to show the construction of the extension trace. For each state s in the queue, it is checked whether s can be blocked by the previous frame F_{d-1} (line 5). If not, a predecessor state t of s that satisfies F_{d-1} is computed and added to the queue (line 7). If a predecessor state is found at level 0, the trace is not extendable and an empty trace is returned. If the state s is blocked at level d, PDRINDGEN, is called to generate a clause that blocks s and possibly others. The clause is then added to all the frames at levels less than or equal to d. The procedure terminates whenever there are no more states to be blocked (or a counterexample was found at line 4). By construction, the output trace $\boldsymbol{G}$ is an extension trace of $\boldsymbol{F}$ at the extension level w. Once PDR extends its trace, PDRPUSH is called to check whether the clauses it learned are also true at higher levels. PDR terminates when the trace is closed.

PDRINDGEN is a crucial optimization to PDR. To block a predecessor s, it is enough to learn the clause $(\neg s)$. However, such a clause is too weak: it cannot block any other predecessors to *Bad*. There could be exponentially many predecessors to a single *Bad* state and thus with this simple strategy, PDR will have to learn exponentially many clauses: one per predecessor. Therefore, it is necessary to be able to block a generalization of the computed predecessor state. Such a generalization must (1) block predecessor s (2) be inductive relative to the previous frame and (3) be satisfied by all initial states . A straightforward method to generate such a generalization is to use the UNSAT core from the relative induction query (line 5) to get rid of parts of state s that are irrelevant to it being blocked. A more aggressive strategy is to assume the generalized clause c when checking for relative induction: $F_i \wedge Tr \wedge c \Rightarrow c'$ [20]. Assuming c at frame i constraints the formula more, making it more likely to be UNSAT. Such an assumption is sound because F_i is just an over-approximation of reachable states. As long as c is satisfied by all initial states, it can be assumed in all frames.

Algorithm 3: PdrBlock.

Input: A transition system $T = (Init, Tr, Bad)$
Input: A SAFE trace $\boldsymbol{F}$ with $|\boldsymbol{F}| = N$
Output: An extension trace $\boldsymbol{G}$ or an empty trace

```
1  w ← N + 1 ; G ← F ; Q.push(⟨Bad, N + 1⟩)
2  while ¬Q.empty() do
3      ⟨s, d⟩ ← Q.pop()
4      if d == 0 then return [ ]
5      if isSat(F_{d−1}(v̄) ∧ Tr(v̄, v̄′) ∧ s(v̄′)) then
6          t ← predecessor(s)
7          Q.push(t, d − 1)
8          Q.push(s, d)
9      else
10         ∀0 ≤ i ≤ d · G_i ← (G_i ∧ PdrIndGen(¬s))
11         w ← min(w, d)
12 return G
```

2.1.5 Avy

Avy, shown in Algorithm 4, is an alternative to Pdr that combines interpolation and recursive blocking. Avy starts with a trace $\boldsymbol{F}$, with $F_0 = Init$, that is extended in every iteration of the main loop. A counterexample is returned whenever $\boldsymbol{F}$ is not extendable (line 3). Otherwise, it calls AvyExtend (Algorithm 5) to extend the trace. It then calls Pdr-Push to check whether lemmas are true at higher frames. Avy converges when the trace is closed.

AvyExtend extends a clausal, monotone, SAFE trace into a stronger clausal, monotone trace. To generate an extension trace for $[F_0, \ldots, F_N]$ at level $0 \leq k \leq N$, it is enough to generate a sequence interpolant $I_{k+1}, \ldots, I_N$ from the unsatisfiability of $Tr[\boldsymbol{F}^k] \wedge Bad(\bar{v}_{N+1})$ and create a trace $[F_0, \ldots F_k, G_{k+1} \ldots, G_N, I_{N+1}]$ by conjoining the corresponding frames: $k < j \leq N \cdot G_j \leftarrow F_j \wedge I_j$. However, the language of the interpolants is not restricted. Therefore, such an extension trace need not be monotone or clausal. To generate a clausal trace, AvyExtend makes use of PdrBlock. Given a SAFE circuit and a clausal trace as input, PdrBlock generates a stronger clausal trace. To make the trace monotone, AvyExtend makes use of the following observation: given a trace $[A_0, \ldots, A_k, A_{k+1}]$ which is monotone until A_k, A_{k+1} can be made monotone if it can be weakened to $A^*_{k+1} \equiv$

$A_{k+1} \vee A_k$.

AvyExtend (Algorithm 5) extracts a sequence interpolant from the unsatisfiability of $Tr[\boldsymbol{F}^k] \wedge Bad(\bar{v}_{N+1})$ (line 1). It then strengthens each element G_{j+1} to the property $P_j = G_j \vee (G_{j+1} \wedge I_{j+1})$ using PdrBlock. Note that all clauses added to frame j are also added to all previous frames to maintain monotonicity. The loop maintains the invariant $G_j \wedge Tr \Rightarrow P_j$. The invariant ensures that G_{j+1} *can* be strengthened. It holds on entry since, from the properties of a trace, $G_j \wedge Tr \Rightarrow G_{j+1}$ and from the properties of an interpolant $G_j \wedge Tr \Rightarrow I_{j+1}$. Let G^*_{j+1} denote the $(j+1)^{th}$ frame after the execution of the loop. Since the invariant holds at the start of the loop, $G^*_{j+1} \Rightarrow P_j$. Therefore, $G^*_{j+1} \wedge Tr \Rightarrow ((G_j \wedge Tr) \vee ((G_{j+1} \wedge Tr) \wedge (I_{j+1} \wedge Tr)))$, or $G^*_{j+1} \wedge Tr \Rightarrow G_{j+1} \vee (G_{j+2} \vee I_{j+2})$, there by proving the invariant for the next iteration of the loop. After every iteration of the loop, PdrPush is called to push the newly learned lemmas forward so that the strengthenings in subsequent iterations are made easier.

Algorithm 4: Avy.

Input: A transition system $T = (Init, Tr, Bad)$

Output: SAFE/UNSAFE

```
1 F_0 ← Init ; N ← 0
2 repeat
3 |   if isSat(Tr[F^0] ∧ Bad(v̄_{N+1})) then return UNSAFE
4 |   k ← max{i | ¬isSat(Tr[F^i] ∧ Bad(v̄_{N+1}))}
5 |   F ← AvyExtend([F_0, ..., F_N], k)
6 |   F ← PdrPush(F)
7 |   if ∃1 ≤ i ≤ N · F_i ⇒ (⋁_{j=0}^{i-1} F_j) then return SAFE
8 |   N ← N + 1
9 until ∞
```

Algorithm 5: AvyExtend.

Input: A clausal, SAFE trace $[F_0, \ldots, F_N]$

Input: An extension level k, s.t. $Tr[\boldsymbol{F}^k] \wedge Bad(\bar{v}_{N+1})$ is unsatisfiable

Output: A clausal, SAFE, extension trace $[G_0, \ldots, G_{N+1}]$

1 $I_{k+1}, \ldots, I_{N+1} \leftarrow$ seqItp$(\,Tr[\boldsymbol{F}^k] \wedge Bad(\bar{v}_{N+1}))$

2 $G \leftarrow [F_0, \ldots, F_N, \top]$

3 **for** $j \leftarrow k$ *to* N *do* **do**

4 $\quad P_j \leftarrow G_j \vee (G_{j+1} \wedge I_{j+1}))$

$\quad$ `// Inv:` $G_j \wedge Tr \Rightarrow P_j$

5 $\quad$ **if** $j == 0$ **then**

6 $\quad\quad [_, _, G_{j+1}] \leftarrow$ PdrBlock$([Init, G_{j+1}], (Init, Tr, \neg(P_j))$

7 $\quad$ **else**

8 $\quad\quad [_, _, G_{j+1}] \leftarrow$ PdrBlock$([Init, G_j, G_{j+1}], (Init, Tr, \neg(P_j))$

9 $\quad \boldsymbol{G} \leftarrow$ PdrPush$(\boldsymbol{G})$

Chapter 3

Separation between strong induction and induction

SAFETY can be established by using either induction or strong induction. For induction, the *certificate for* SAFETY is a 1-inductive invariant. For strong induction, the *certificate for* SAFETY is a k-inductive invariant, for some arbitrary k. For proving SAFETY in propositional logic, induction and strong induction have the same deductive power: if a system admits a k-inductive invariant, it also necessarily admits a 1-inductive invariant [46]. However, it is conjectured that there exists an exponential separation between the sizes of the minimal k-inductive, and minimal 1-inductive invariants [17]. This makes it seem that generating k-inductive invariants is much more efficient that generating 1-inductive invariants. While this is true, *verifying* a k-inductive invariant is as hard as generating a 1-inductive invariant: given a proof of k-induction, we can generate a 1-inductive invariant of the same size (see Section 4.1). In this chapter, we give a constructive proof of separation between sizes of the minimal k-inductive invariant, and minimal 1-inductive invariant. We also discuss various factors affecting algorithms driven by strong induction and induction.

We study the relationship between induction and strong induction by constructing a family of transition systems parameterized by the number of variables n, such that the minimal k-inductive invariant is of constant size whereas the minimal inductive invariant grows exponentially with n. The size of a formula can have multiple definitions. We are concerned with the size of a formula when written in CNF:

Definition 2. *The size of a formula f, denoted by $|f|$, is the number of clauses in the minimal CNF representation of f.*

A SAFE transition system can have multiple SAFE inductive invariants. Since we want to establish a separation, we are concerned with the minimal SAFE inductive invariant. Let 1-$ind(T)$ denote the *minimal* SAFE 1-inductive invariant for the SAFE transition system T. Similarly, let 2-$ind(T)$ denote the *minimal* SAFE 2-inductive invariant for the SAFE transition system T.

We first show a separation between 2-inductive invariants and 1-inductive invariants (Section 3.1). While this is enough to prove a separation between certificates for strong induction and induction, we go one step further and generalize such systems (Section 3.2).

3.1 Separation between 1-induction and 2-induction

Let $f(b_1, b_2, \ldots, b_n)$ be a propositional formula with variables $b_1, b_2, \ldots, b_n$. For simplicity, we write f_n to mean $f(b_1, b_2, \ldots, b_n)$ and f'_n to mean f_n over primed variables.

We construct a *counter*[1] circuit using f_n as follows:

$$\begin{aligned} C_{f_n} &= (\bar{v}, Init_{f_n}, Tr_{f_n}, Bad) \\ \bar{v} &= a, b_1, b_2, \ldots, b_n \\ Tr_{f_n} &= (f_n \Rightarrow f'_n) \land (a \Rightarrow (a' \Leftrightarrow f'_n)) \\ Init_{f_n} &= a \land f_n \\ Bad &= \neg a \end{aligned}$$

Figure 3.1 shows the states and transitions that the counter circuit can have. All states of the circuit satisfy one of three formulas: *Init*, *Bad* or $a \land \neg f_n$. From a state that satisfies $a \land \neg f_n$, the circuit can either transition to a *Bad* state or a non-bad state. We can see that an initial state will always transition into another "initial" state: a state satisfying $Init_{f_n}$. That is, $Init_{f_n}$ is closed under an application of Tr_{f_n}. Therefore, the set of all reachable states in the system is $Init_{f_n}$. Clearly C_{f_n} is SAFE.

Lemma 2. *All* SAFE *inductive invariants of the counter circuit* C_{f_n} *are equivalent to* $Init_{f_n}$.

Proof. Since $Init_{f_n}$ is exactly the set of all reachable states, it is an inductive invariant. Let *Inv* be an inductive invariant. We are going to prove that $Inv \equiv Init_{f_n}$. By the properties

[1] This circuit and its generalizations behave very similar to a ring counter. In conditions of interest, the next state is a permutation of the previous state.

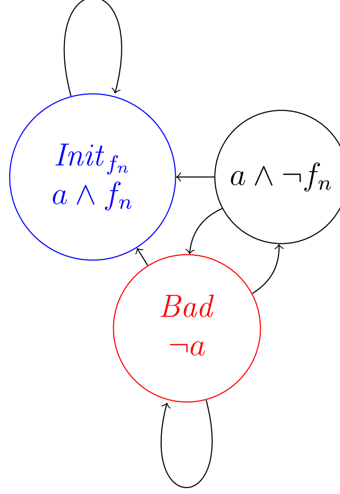


Figure 3.1: State diagram for the counter circuit after labelling them $Init_{f_n}$, Bad or $a \wedge f_n$. States satisfying Bad can transition into states in any of the three categories. However, a state satisfying $Init_{f_n}$ can only transition into one satisfying $Init_{f_n}$.

of an inductive invariant, $Init_{f_n}$ has to be stronger than Inv, that is all initial states satisfy Inv. We prove that all states that satisfy Inv also satisfy $Init_{f_n}$.

Since Inv is SAFE, $Inv \Rightarrow a$. That is, any state s that satisfies Inv assigns a to true. From Tr_{f_n}, any successor state t of s must satisfy $a \Leftrightarrow f_n$. Since Inv is closed under application of Tr_{f_n}, t should also satisfy Inv. That is, $Inv \Rightarrow a \Leftrightarrow f_n$. Therefore $Inv \Rightarrow a \wedge f_n$. Thus all states that satisfy Inv also satisfy $Init_{f_n}$. Therefore, $Inv \equiv Init_{f_n}$.

□

Corollary 1. $|$ *1-ind*(C_{f_n}) $| = |\,(a \wedge f_n)\,|$.

Thus, the size of the 1-inductive invariant is dependent on the formula f_n.

Lemma 3. *a is a* SAFE *2-inductive invariant for the counter circuit C_{f_n}.*

Proof. It is clear that a satisfies initiation (Equation 2.3) and proves safety (Equation 2.5). Therefore, a is a SAFE 2-inductive invariant if it is 2-inductive: $a \wedge Tr_{f_n} \wedge a'' \wedge Tr_{f'_n} \Rightarrow a''$.

Note that 2-induction allows us to assume a'. We can deduce the following:

$$\begin{aligned}
a \wedge Tr_{f_n} &\Rightarrow (a' \Leftrightarrow f'_n) \\
(a' \Leftrightarrow f'_n) \wedge a' &\Rightarrow f'_n \\
f'_n \wedge Tr_{f_n} &\Rightarrow f''_n \\
a' \wedge Tr_{f_n} &\Rightarrow (a'' \Leftrightarrow f''_n) \\
(a'' \Leftrightarrow f''_n) \wedge f''_n &\Rightarrow a''
\end{aligned}$$

□

Corollary 2. $\mid$ *2-ind*$(C_{f(n)}) \mid = |a|$.

Corollary 1 shows that the size of the minimal one inductive invariant is dependent on the size of f_n. Thus, to prove a separation, it is enough to construct a counter circuit with a suitable f_n. Let xor be the parity function $xor(b_1, b_2, \ldots, b_n) = \sum_i b_i \mod 2$. xor does not have a small CNF representation: $|xor_n| = 2^{n-1}$ [73]. Thus, constructing a counter circuit with xor gives us an exponential separation:

The minimal 1-inductive invariant is exponentially larger than the minimal 2-inductive invariant for the counter circuit using xor:

Theorem 2. $\mid$ *1-ind*$(C_{xor_n}) \mid = 2^{n-1} \mid$ *2-ind*$(C_{xor_n}) \mid$.

Theorem 2 uses Definition 2 for the *size* of a boolean formula. By choosing a different measure for size and a different formula, the theorem can be generalized further. If we choose a basic set of gates (called a basis), we can define the *circuit size* of a formula to be the number of basis elements used to construct a circuit for the formula. Circuit complexity tells us that there are functions whose circuits are of exponential size in any basis with bounded fan-in [70]. That is, we can prove a much stronger result: there exists a function f such that the *circuit size* of 1-$ind(C_{f_n})$ is exponentially larger than that of 2-$ind(C_{f_n})$. Unfortunately, such an f has not been constructed yet [68].

3.1.1 Validating certificates for strong induction and induction

1-inductive invariants and 2-inductive invariants are certificates for the SAFETY. To verify the certificates, we need to prove their validity . That is, we need to show that they satisfy initiation, induction and are SAFE. The most common proof system used in propositional

logic is general resolution (or simply resolution). Thus, to compare *proofs* of SAFETY, we need to construct and compare resolution proofs for the certificates.

Consider proving the validity of the inductive step. To validate a 1-inductive invariant Inv, it is sufficient to prove the unsatisfiability of $Inv \wedge Tr \wedge \neg Inv'$. Since $a \wedge xor_n$ is the certificate for 1-induction, validating it is equivalent to proving the unsatisfiability of $a \wedge xor_n \wedge Tr_{xor_n} \wedge (\neg a' \vee \overline{xor}_n)$. This necessarily involves resolving each clause in xor_n with each clause in $(\overline{xor}_n \vee xor'_n)$, leading to 2^{n+1} resolution steps. A constant number of such big resolution steps are required to derive $\perp$, leading to a proof size of $\Theta(2^n)$.

To validate a 2-inductive invariant Inv, it is sufficient to prove the unsatisfiability of $Inv \wedge Tr \wedge Inv' \wedge Tr \wedge \neg Inv''$. Since a is the 2-inductive invariant, we need to prove the unsatisfiability of $a \wedge Tr_{xor_n} \wedge a' \wedge Tr_{xor'_n} \wedge \neg a''$. This also necessarily involves resolving each clause in xor_n with each clause in $(\overline{xor}_n \vee xor'_n)$. Thus, the size of this resolution proof is $\Theta(2^n)$ as well.

For initiation, the size of proof for 1-induction is $\Theta(n)$ and that for 2-induction is $\Theta(1)$. For safety, the size of proofs are $\Theta(1)$ in both cases.

We observe that even though the 1-inductive invariant is exponentially larger than the 2-inductive invariant, the resolution proofs for the certificates are of the same size.

3.1.2 PDR vs strong induction on C_{xor_n}

PDR (Section 2.1.4) constructs a 1-inductive invariant whereas strong induction (Section 2.1.1) constructs a k-inductive invariant for arbitrary k. Since the only 1-inductive invariant is of size $\Theta(2^n)$, PDR will make at least 2^n SAT queries before converging. The size of each of these queries will also be $\Theta(2^n)$. Whereas, the strong induction would only require 3 queries (one for checking instantiation, one for checking 1-induction and one for checking 2-induction). Each query will be of size $\Theta(2^n)$. However, this need not result in a massive difference in running times since the resolution proofs are of the same size.

3.2 Generalization of the counter circuit

In this section, we discuss how to construct systems that will separate an l-inductive invariant from a 1-inductive invariant for an arbitrary but fixed l. We will see that the counter circuit from previous section was an instantiation ($l = 2$) of a family of circuits for which there exists a separation between certificates for induction and strong induction.

Let $F = \{f_{1n}, f_{2n}, \ldots, f_{ln}\}$ be a set of l formulas, each with n variables. For simplicity, we write f_{jn} to mean $f_j(b_1, b_2, \ldots, b_n)$ and f'_{jn} to mean f_{jn} over primed variables. We construct a counter circuit with F as follows:

$$
\begin{aligned}
C_F =& (\bar{v}, \mathit{Init}_F, \mathit{Tr}_F, \mathit{Bad}) \\
\bar{v} =& a, b_1, b_2, \ldots, b_n \\
\mathit{Tr}_F =& (f_{1n} \Rightarrow f'_{2n}) \wedge (f_{2n} \Rightarrow f'_{3n}) \wedge \ldots \wedge (f_{ln} \Rightarrow f'_{1n}) \wedge \\
& a \Rightarrow (a' \Leftrightarrow f'_{1n}) \\
\mathit{Init}_F =& f_{1n} \wedge f_{2n} \wedge \ldots \wedge f_{ln} \wedge a \\
\mathit{Bad} =& \neg a
\end{aligned}
$$

The set of all reachable states is Init_F itself. a is an l-inductive invariant. By assuming a to be true for l steps, the left hand side of Tr_F becomes true and implies a in the next step.

An inductive invariant is Init_F. Let Inv be an inductive invariant that proves SAFETY. We are going to prove that $\mathit{Inv} \equiv \mathit{Init}_F$. The right to left direction is a property of any inductive invariant. Since $\mathit{Inv} \Rightarrow a$, $\mathit{Inv} \wedge \mathit{Tr}_F \Rightarrow a' \Leftrightarrow f'_{1n}$. Since inductive invariants are closed under transition, $\mathit{Inv} \Rightarrow a \Leftrightarrow f_{1n}$. From this it follows that $\mathit{Inv} \Rightarrow f_{1n}$. Therefore $\mathit{Inv} \wedge \mathit{Tr}_F \Rightarrow f'_{2n}$. By the same argument, $\mathit{Inv} \Rightarrow f_{2n}$. In the same way, $\mathit{Inv} \Rightarrow a \wedge f_{1n} \wedge f_{2n} \wedge \ldots \wedge f_{ln}$. Thus, all inductive invariants are equivalent to Init_F. Therefore, if Init_F does not have a small CNF representation, neither does any of the 1-inductive invariants.

We can construct the formulas in F by taking a large CNF formula and splitting it arbitrarily into l disjoint sub formulas, taking care that none of them are VALID or UNSAT.

To summarize, this chapter showed that for the *counter* circuits made out of *xor* functions, the minimal 1-inductive invariant is exponentially larger than the minimal 2-inductive invariant. This proves that the certificates for 1-induction could be exponentially larger than certificates for strong induction. Thus, in a guess-and-check approach to find inductive invariants, using the principle of strong induction would lead to guessing smaller invariants at no additional cost to checking.

Chapter 4

Kavy

In this chapter, we present kAvy, an SMC algorithm that uses the principle of strong induction to extend an inductive trace. The chapter is structured as follows. First, we introduce the concept of extending a trace using relative strong induction. Second, we present kAvy and describe the details of how strong induction is used to compute an extended trace. Third, we describe two techniques for computing maximal parameters to apply strong induction. Unless stated otherwise, we assume that all traces are monotone.

Definition 3. *A safe trace $\boldsymbol{F}$, with $|\boldsymbol{F}| = N$, is **strongly extendable** with respect to (i, k), where $1 \leq k \leq i+1 \leq N+1$, iff there exists a safe inductive trace $\boldsymbol{G}$ stronger than $\boldsymbol{F}$ such that $|\boldsymbol{G}| > N$ and $Tr[F_i]^k \Rightarrow G_{i+1}$.*

We refer to the pair (i, k) as *a strong extension level (SEL)*, and to the trace $\boldsymbol{G}$ as an *(i, k)-extension trace*, or simply a *strong extension trace (SET)* when (i, k) is not important. Note that for $k = 1$, $\boldsymbol{G}$ is just an extension trace.

Example 2. *For Fig. 1.1, the trace $\boldsymbol{F} = [c = 0, c < 66]$ is strongly extendable at level 1. A valid $(1, 2)$-externsion trace is $\boldsymbol{G} = [c = 0, (c \neq 65) \wedge (c < 66), c < 66]$. Note that $(c < 66)$ is 2-inductive relative to F_1, i.e. $Tr[F_1]^2 \Rightarrow (c'' < 66)$.*

We write $\mathcal{K}(\boldsymbol{F})$ for the set of all SELs of $\boldsymbol{F}$. We define an order on SELs by : $(i_1, k_1) \preceq (i_2, k_2)$ iff (i) $i_1 < i_2$; or (ii) $i_1 = i_2 \wedge k_1 > k_2$. The maximal SEL is $\max(\mathcal{K}(\boldsymbol{F}))$. Note that the existence of a SEL (i, k) means that an unrolling of the i-suffix with F_i repeated k times does not contain any bad states. We use $Tr[\![\boldsymbol{F}^i]\!]^k$ to denote this *characteristic formula* for SEL (i, k) :

$$Tr[\![\boldsymbol{F}^i]\!]^k = \begin{cases} Tr[F_i]_{i+1-k}^{i+1} \wedge Tr[\boldsymbol{F}^{i+1}] & \text{if } 0 \leq i < N \\ Tr[F_N]_{N+1-k}^{N+1} & \text{if } i = N \end{cases} \tag{4.1}$$

Lemma 4. *Let $\boldsymbol{F}$ be a safe trace, where $|\boldsymbol{F}| = N$. Then, (i, k), $1 \leq k \leq i + 1 \leq N + 1$, is an SEL of $\boldsymbol{F}$ iff the formula $Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1})$ is unsatisfiable.*

Proof. kAvyExtend (Algorithm 7) proves the right-to-left direction by constructing an (i, k)-extension trace from the unsatisfiability of $Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1})$. We prove the left-to-right direction using contradiction. Let $\boldsymbol{G}$ be an (i, k)-extension trace of $\boldsymbol{F}$. Since $\boldsymbol{G}$ is SAFE upto $(N + 1)$, G_{i+1} is strong enough to block all *predecessor sequences* of length $(N + 1 - i)$. We show that if $Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1})$ is satisfiable, G_{i+1} admits a predecessor sequence, there by arriving at a contradiction.

If $Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1})$ is satisfiable, so is the weaker formula $Tr[F_i]_{i-k}^i \wedge Tr_i^{N+1} \wedge Bad(\bar{v}_{N+1})$. Let $s_i^1, s_i^2, \ldots, s_i^k, s_{i+1}, \ldots s_{N+1}$ be a sequence of states that satisfies the weaker formula. All s_i's satisfy F_i. Since $Tr[F_i]^k \Rightarrow G_{i+1}$, s_{i+1} necessarily satisfies G_{i+1}. That is $s_{i+1}, s_{i+2}, \ldots s_{N+1}$ is an $(N + 1 - i)$ length predecessor sequence not blocked by G_{i+1}. This contradicts the claim that G_{i+1} is strong enough to block any such predecessor sequence. Hence $Tr[F_i]_{i-k}^i \wedge Tr_i^{N+1} \wedge Bad(\bar{v}_{N+1})$ is unsatisfiable. Therefore the stronger formula $Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1})$ is also unsatisfiable.

□

Notice that the unsatisfiability is caused due to F_i. Not all the frames in $\boldsymbol{F}^i$ are necessary. However, as we will see, the suffix will make it easier to extend the trace.

The level i in the maximal SEL (i, k) of a given trace $\boldsymbol{F}$ is greater or equal to the maximal extension level of $\boldsymbol{F}$:

Lemma 5. *Let $(i, k) = \max(\mathcal{K}(\boldsymbol{F}))$, then $i \geq \max(\mathcal{W}(\boldsymbol{F}))$.*

Hence, extensions based on maximal SEL are constructed from frames at higher level compared to extensions based on maximal extension level.

Example 3. *For Fig. 1.1, the trace $[c = 0, c < 66]$ has a maximum extension level of 0. Since $(c < 66)$ is 2-inductive, the trace is strongly extendable at level 1 (as was seen in Example 2).*

kAvy Algorithm

kAvy is shown in Fig. 6. It starts with an inductive trace $\boldsymbol{F} = [Init]$ and iteratively extends $\boldsymbol{F}$ using SELs. A counterexample is returned if the trace cannot be extended (line 4). Otherwise, kAvy computes the largest extension level (line 5) (described in Section 4.2).

Algorithm 6: kAvy algorithm.

Input: A transition system $T = (Init, Tr, Bad)$
Output: SAFE/UNSAFE
1 $\boldsymbol{F} \leftarrow [Init]$; $N \leftarrow 0$
2 **repeat**
   // Invariant: $\boldsymbol{F}$ is a monotone, clausal, safe, inductive trace
3   $U \leftarrow Tr[\boldsymbol{F}^0] \wedge Bad(\bar{v}_{N+1})$
4   **if** isSat(U) **then return** UNSAFE
5   $(i,k) \leftarrow \max\{(i,k) \mid \neg\text{isSat}(Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1}))\}$
6   $[F_0, \dots, F_{N+1}] \leftarrow$ kAvyExtend($\boldsymbol{F}, (i,k)$)
7   $[F_0, \dots, F_{N+1}] \leftarrow$ PdrPush($[F_0, \dots, F_{N+1}]$)
8   **if** $\exists 1 \leq i \leq N \cdot F_i \Rightarrow \left(\bigvee_{j=0}^{i-1} F_j\right)$ **then return** SAFE
9   $N \leftarrow N + 1$
10 **until** ∞

Then, it constructs a strong extension trace using kAvyExtend (line 6) (described in Section 4.1). Finally, PdrPush is called to check whether the trace is closed. Note that $\boldsymbol{F}$ is a monotone, clausal, safe inductive trace throughout the algorithm.

4.1 Extending a trace with strong induction

In this section, we describe the procedure kAvyExtend (shown in Algorithm 7) that, given a trace $\boldsymbol{F}$ of size $|\boldsymbol{F}| = N$ and an (i,k) SEL of $\boldsymbol{F}$ constructs an (i,k)-extension trace $\boldsymbol{G}$ of size $|\boldsymbol{G}| = N + 1$. The procedure itself is fairly simple, but its proof of correctness is complex. We first present the theoretical results that connect sequence interpolants with strong extension traces, then the procedure, and then details of its correctness. Through the section, we fix a trace $\boldsymbol{F}$ and its SEL (i,k).

Sequence interpolation for SEL. Let (i,k) be an SEL of $\boldsymbol{F}$. By Lemma 4, $\Psi = Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1})$ is unsatisfiable. Let $\mathcal{A} = \{A_{i-k+1}, \dots, A_{N+1}\}$ be a partitioning of Ψ defined as follows:

$$A_j = \begin{cases} F_i(\bar{v}_j) \wedge Tr(\bar{v}_j, \bar{v}_{j+1}) & \text{if } i-k+1 \leq j \leq i \\ F_j(\bar{v}_j) \wedge Tr(\bar{v}_j, \bar{v}_{j+1}) & \text{if } i < j \leq N \\ Bad(\bar{v}_{N+1}) & \text{if } j = N+1 \end{cases}$$

Since $(\wedge\mathcal{A}) = \Psi$, $\mathcal{A}$ is unsatisfiable. Let $\boldsymbol{I} = [I_{i-k+2}, \ldots, I_{N+1}]$ be a sequence interpolant corresponding to $\mathcal{A}$. Then, $\boldsymbol{I}$ satisfies the following properties:

$$\begin{aligned} F_i \wedge Tr &\Rightarrow I'_{i-k+2} & \quad \forall i-k+2 \le j \le i \cdot (F_i \wedge I_j) \wedge Tr &\Rightarrow I'_{j+1} \\ I_{N+1} &\Rightarrow \neg Bad & \forall i < j \le N \cdot (F_j \wedge I_j) \wedge Tr &\Rightarrow I'_{j+1} \end{aligned} \qquad (\clubsuit)$$

Note that in ($\clubsuit$), both i and k are fixed — they are the (i, k)-extension level. Furthermore, in the top row F_i is fixed as well.

The conjunction of the first k interpolants in $\boldsymbol{I}$ is k-inductive relative to the frame F_i:

Lemma 6. *The formula* $F_{i+1} \wedge \left(\bigwedge_{m=i-k+2}^{i+1} I_m\right)$ *is* k*-inductive relative to* F_i*.*

Proof. Since F_i and F_{i+1} are consecutive frames of a trace, $F_i \wedge Tr \Rightarrow F'_{i+1}$. Thus, $\forall i - k + 2 \le j \le i \cdot Tr[F_i]^j_{i-k+2} \Rightarrow F_{i+1}(\bar{v}_{j+1})$. Moreover, by ($\clubsuit$), $F_i \wedge Tr \Rightarrow I'_{i-k+2}$ and $\forall i-k+2 \le j \le i+1 \cdot (F_i \wedge I_j) \wedge Tr \Rightarrow I'_{j+1}$. Equivalently, $\forall i-k+2 \le j \le i+1 \cdot Tr[F_i]^j_{i-k+2} \Rightarrow I_{j+1}(\bar{v}_{j+1})$. By induction over the difference between $(i+1)$ and $(i-k+2)$, we show that $Tr[F_i]^{i+1}_{i-k+2} \Rightarrow (F_{i+1} \wedge \bigwedge_{m=i-k+2}^{i+1} I_m)(\bar{v}_{i+1})$, which concludes the proof. □

We use Lemma 6 to construct a strong extension trace $\boldsymbol{G}$:

Lemma 7. *Let* $\boldsymbol{G} = [G_0, \ldots, G_{N+1}]$*, be an inductive trace defined as follows:*

$$G_j = \begin{cases} F_j & \text{if } 0 \le j < i-k+2 \\ F_j \wedge \left(\bigwedge_{m=i-k+2}^{j} I_m\right) & \text{if } i-k+2 \le j < i+2 \\ (F_j \wedge I_j) & \text{if } i+2 \le j < N+1 \\ I_{N+1} & \text{if } j = (N+1) \end{cases}$$

Then, $\boldsymbol{G}$ *is an* (i, k)*-extension trace of* $\boldsymbol{F}$ *(not necessarily monotone).*

Proof. By Lemma 6, G_{i+1} is k-inductive relative to F_i. Therefore, it is sufficient to show that $\boldsymbol{G}$ is a safe inductive trace that is stronger than $\boldsymbol{F}$. By definition, $\forall 0 \le j \le N \cdot G_j \Rightarrow F_j$. By ($\clubsuit$), $F_i \wedge Tr \Rightarrow I'_{i-k+2}$ and $\forall i-k+2 \le j < i+2 \cdot (F_i \wedge I_j) \wedge Tr \Rightarrow I'_{j+1}$. By induction over j, $\left((F_i \wedge \bigwedge_{m=i-k+2}^{j} I_m) \wedge Tr\right) \Rightarrow \bigwedge_{m=i-k+2}^{j+1} I'_m$ for all $i-k+2 \le j < i+2$. Since $\boldsymbol{F}$ is monotone, $\forall i-k+2 \le j < i+2 \cdot \left((F_j \wedge \bigwedge_{m=i-k+2}^{j} I_m) \wedge Tr\right) \Rightarrow \bigwedge_{m=i-k+2}^{j+1} I'_m$

By ($\clubsuit$), $\forall i < j \le N \cdot (F_j \wedge I_j) \wedge Tr \Rightarrow I'_{j+1}$. Again, since $\boldsymbol{F}$ is a trace, we conclude that $\forall i < j < N \cdot (F_j \wedge I_j) \wedge Tr \Rightarrow (F_{j+1} \wedge I_{j+1})'$. Combining the above, $G_j \wedge Tr \Rightarrow G'_{j+1}$ for $0 \le j \le N$. Since $\boldsymbol{F}$ is safe and $I_{N+1} \Rightarrow \neg Bad$, then $\boldsymbol{G}$ is safe and stronger than $\boldsymbol{F}$. □

Lemma 7 defines an obvious procedure to construct an (i,k)-extension trace $\boldsymbol{G}$ for $\boldsymbol{F}$. However, such $\boldsymbol{G}$ is neither monotone nor clausal. In the rest of this section, we describe the procedure kAvyExtend that starts with a sequence interpolant (as in Lemma 7), but uses PdrBlock to systematically construct a safe monotone clausal extension of $\boldsymbol{F}$.

Algorithm 7: kAvyExtend. The invariants marked † hold only when the PdrBlock does no inductive generalization.

Input: a monotone, clausal, safe trace $\boldsymbol{F}$ of size N
Input: A strong extension level (i,k) s.t. $Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1})$ is unsatisfiable
Output: a monotone, clausal, safe trace $\boldsymbol{G}$ of size $N+1$

1 $I_{i-k+2}, \ldots, I_{N+1} \leftarrow$ seqItp$(Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1}))$
2 $\boldsymbol{G} \leftarrow [F_0, \ldots, F_N, \top]$
3 **for** $j \leftarrow i-k+1$ **to** i **do**
4 $\quad P_j \leftarrow (G_j \vee (G_{i+1} \wedge I_{j+1}))$
$\quad$ // Inv$_1$: $\boldsymbol{G}$ is monotone and clausal
$\quad$ // Inv$_2$: $G_i \wedge Tr \Rightarrow P_j$
$\quad$ // Inv$_3^\dagger$: $\forall j < m \leq (i+1) \cdot G_m \equiv F_m \wedge \bigwedge_{\ell=i-k+1}^{j-1} (G_\ell \vee I_{\ell+1})$
$\quad$ // Inv$_3$: $\forall j < m \leq (i+1) \cdot G_m \Rightarrow F_m \wedge \bigwedge_{\ell=i-k+1}^{j-1} (G_\ell \vee I_{\ell+1})$
5 $\quad [_, _, G_{i+1}] \leftarrow$ PdrBlock$([Init, G_i, G_{i+1}], (Init, Tr, \neg P_j))$
6 $P_i \leftarrow (G_i \vee (G_{i+1} \wedge I_{j+1}))$
7 **if** $i = 0$ **then** $[_, _, G_{i+1}] \leftarrow$ PdrBlock$([Init, G_{i+1}], (Init, Tr, \neg P_i))$
8 **else** $[_, _, G_{i+1}] \leftarrow$ PdrBlock$([Init, G_i, G_{i+1}], (Init, Tr, \neg P_i))$
// Inv$_4^\dagger$: $G_{i+1} \equiv F_{i+1} \wedge \bigwedge_{\ell=i-k+1}^{i} (G_\ell \vee I_{\ell+1})$
// Inv$_4$: $G_{i+1} \Rightarrow F_{i+1} \wedge \bigwedge_{\ell=i-k+1}^{i} (G_\ell \vee I_{\ell+1})$
9 **for** $j \leftarrow i+1$ **to** $N+1$ **do**
10 $\quad P_j \leftarrow G_j \vee (G_{j+1} \wedge I_{j+1})$
$\quad$ // Inv$_6$: $G_j \wedge Tr \Rightarrow P_j$
11 $\quad [_, _, G_{j+1}] \leftarrow$ PdrBlock$([Init, G_j, G_{j+1}], (Init, Tr, \neg P_j))$
12 $\quad \boldsymbol{G} \leftarrow$ PdrPush$(\boldsymbol{G})$
// Inv$_7^\dagger$: $\boldsymbol{G}$ is an (i,k)-extension trace of $\boldsymbol{F}$
// Inv$_7$: $\boldsymbol{G}$ is an extension trace of $\boldsymbol{F}$
13 **return** $\boldsymbol{G}$

The procedure kAvyExtend is shown in Algorithm 7. For simplicity of the presentation, we assume that PdrBlock does not use inductive generalization. The invariants marked by † rely on this assumption. We stress that the assumption is for presentation

only. The correctness of kAvyExtend is independent of it.

kAvyExtend starts with a sequence interpolant according to the partitioning $\mathcal{A}$. The extension trace $\boldsymbol{G}$ is initialized to $\boldsymbol{F}$ and G_{N+1} is initialized to $\top$ (line 2). The rest proceeds in three phases: *Phase 1* (lines 3–5) computes the prefix $G_{i-k+2}, \ldots, G_{i+1}$ using the first $k-1$ elements of $\boldsymbol{I}$; *Phase 2* (line 8) computes G_{i+1} using I_{i+1}; *Phase 3* (lines 9–12) computes the suffix $\boldsymbol{G}^{i+2}$ using the last $(N-i)$ elements of $\boldsymbol{I}$. During this phase, PdrPush (line 12) pushes clauses forward so that they can be used in the next iteration. The correctness of the phases follows from the invariants shown in Alg. 7. We present each phase in turn.

Recall that PdrBlock takes a trace $\boldsymbol{F}$ (that is safe up to the last frame) and a transition system, and returns a safe strengthening of $\boldsymbol{F}$, while ensuring that the result is monotone and clausal. This guarantee is maintained by Algorithm 7, by requiring that any clause added to any frame G_i of $\boldsymbol{G}$ is implicitly added to all frames below G_i.

Phase 1. By Lemma 6, the first k elements of the sequence interpolant computed at line 1 over-approximate states reachable in $i+1$ steps of Tr. Phase 1 uses this to strengthen G_{i+1} using the first k elements of $\boldsymbol{I}$. Note that in that phase, new clauses are always added to frame G_{i+1}, and all frames before it!

Correctness of Phase 1 (line 5) follows from the loop invariant $\texttt{Inv}_2$. It holds on loop entry since $G_i \wedge Tr \Rightarrow I_{i-k+2}$ (since $G_i = F_i$ and (♣)) and $G_i \wedge Tr \Rightarrow G_{i+1}$ (since $\boldsymbol{G}$ is initially a trace). Let G_i and G_i^* be the ith frame before and after execution of iteration j of the loop, respectively. PdrBlock blocks $\neg P_j$ at iteration j of the loop. Assume that $\texttt{Inv}_2$ holds at the beginning of the loop. Then, $G_i^* \Rightarrow G_i \wedge P_j$ since PdrBlock strengthens G_i. Since $G_j \Rightarrow G_i$ and $G_i \Rightarrow G_{i+1}$, this simplifies to $G_i^* \Rightarrow G_j \vee (G_i \wedge I_{j+1})$. Finally, since $\boldsymbol{G}$ is a trace, $\texttt{Inv}_2$ holds at the end of the iteration.

$\texttt{Inv}_2$ ensures that the trace given to PdrBlock at line 5 *can* be made safe relative to P_j. From the post-condition of PdrBlock, it follows that at iteration j, G_{i+1} is strengthened to G_{i+1}^* such that $G_{i+1}^* \Rightarrow P_j$ and $\boldsymbol{G}$ remains a monotone clausal trace. At the end of *Phase 1*, $[G_0, \ldots, G_{i+1}]$ is a clausal monotone trace.

Interestingly, the calls to PdrBlock in this phase do not satisfy an expected pre-condition: the frame G_i in $[Init, G_i, G_{i+1}]$ might not be safe for property P_j. However, we can see that $Init \Rightarrow P_j$ and from $\texttt{Inv}_2$, it is clear that P_j is inductive relative to G_i. This is a sufficient precondition for PdrBlock.

Phase 2. This phase strengthens G_{i+1} using the interpolant I_{i+1}. After Phase 2, G_{i+1} is k-inductive relative to F_i.

Phase 3. Unlike *Phase 1*, G_{j+1} is computed at the jth iteration. Because of this, the property P_j in this phase is slightly different than that of Phase 1. Correctness follows from invariant $\texttt{Inv}_6$ that ensures that at iteration j, G_{j+1} *can* be made safe relative to P_j. From the post-condition of PdrBlock, it follows that G_{j+1} is strengthened to G^*_{j+1} such that $G^*_{j+1} \Rightarrow P_j$ and $\boldsymbol{G}$ is a monotone clausal trace. The invariant implies that at the end of the loop $G_{N+1} \Rightarrow G_N \vee I_{N+1}$, making $\boldsymbol{G}$ safe. Thus, at the end of the loop $\boldsymbol{G}$ is a safe monotone clausal trace that is stronger than $\boldsymbol{F}$. What remains is to show is that G_{i+1} is k-inductive relative to F_i.

Let φ be the formula from Lemma 6. Assuming that PdrBlock did no inductive generalization, *Phase 1* maintains $\texttt{Inv}^\dagger_3$, which states that at iteration j, PdrBlock strengthens frames $\{G_m\}$, $j < m \leq (i+1)$. $\texttt{Inv}^\dagger_3$ holds on loop entry, since initially $\boldsymbol{G} = \boldsymbol{F}$. Let G_m, G^*_m ($j < m \leq (i+1)$) be frame m at the beginning and at the end of the loop iteration, respectively. In the loop, PdrBlock adds clauses that block $\neg P_j$. Thus, $G^*_m \equiv G_m \wedge P_j$. Since $G_j \Rightarrow G_m$, this simplifies to $G^*_m \equiv G_m \wedge (G_j \vee I_{j+1})$. Expanding G_m, we get $G^*_m \equiv F_m \wedge \bigwedge_{\ell=i-k+1}^{j} (G_\ell \vee I_{\ell+1})$. Thus, $\texttt{Inv}^\dagger_3$ holds at the end of the loop.

In particular, after line 8, $G_{i+1} \equiv F_{i+1} \wedge \bigwedge_{\ell=i-k+1}^{i} (G_\ell \vee I_{\ell+1})$. Since $\varphi \Rightarrow G_{i+1}$, G_{i+1} is k-inductive relative to F_i.

Theorem 3. *Given a safe trace $\boldsymbol{F}$ of size N and an SEL (i,k) for $\boldsymbol{F}$,* kAvyExtend *returns a clausal monotone extension trace $\boldsymbol{G}$ of size $N+1$. Furthermore, if* PdrBlock *does no inductive generalization then $\boldsymbol{G}$ is an (i,k)-extension trace.*

Of course, assuming that PdrBlock does no inductive generalization is not realistic. kAvyExtend remains correct without the assumption: it returns a trace $\boldsymbol{G}$ that is a monotone clausal extension of $\boldsymbol{F}$. However, $\boldsymbol{G}$ might be stronger than any (i,k)-extension of $\boldsymbol{F}$. The invariants marked with † are then relaxed to their unmarked versions. Overall, inductive generalization improves kAvyExtend since it is not restricted to only a k-inductive strengthening.

Importantly, the output of kAvyExtend is a regular inductive trace. Thus, kAvyExtend is a procedure to strengthen a (relatively) k-inductive certificate to a (relatively) 1-inductive certificate. Hence, after kAvyExtend, any strategy for further generalization or trace extension from IC3, Pdr, or Avy is applicable.

4.2 Searching for the maximal SEL

In this section, we describe two algorithms for computing the maximal SEL. Both algorithms can be used to implement line 5 of Alg. 6. They perform a guided search for group minimal unsatisfiable subsets. They terminate when having fewer clauses would not increase the SEL further. The first, called *top-down*, starts from the largest unrolling of the Tr and then reduces the length of the unrolling. The second, called *bottom-up*, finds the largest (regular) extension level first, and then grows it using strong induction.

Top-down SEL. A pair (i, k) is the maximal SEL iff

$$i = \max \{j \mid 0 \leq j \leq N \cdot Tr[\![\boldsymbol{F}^j]\!]^{j+1} \wedge Bad(\bar{v}_{N+1}) \Rightarrow \bot\}$$
$$k = \min \{\ell \mid 1 \leq \ell \leq (i+1) \cdot Tr[\![\boldsymbol{F}^i]\!]^{\ell} \wedge Bad(\bar{v}_{N+1}) \Rightarrow \bot\}$$

Note that k depends on i. For a SEL $(i, k) \in \mathcal{K}(\boldsymbol{F})$, we refer to the formula $Tr[\boldsymbol{F}^i]$ as a *suffix* and to number k as the depth of induction. Thus, the search can be split into two phases: (a) find the smallest suffix while using the maximal depth of induction allowed (for that suffix), and (b) minimizing the depth of induction k for the value of i found in step (a). This is captured in Alg. 8. The algorithm requires at most $(N + 1)$ SAT queries. One downside, however, is that the formulas constructed in the first phase (line 3) are large because the depth of induction is the maximum possible.

Algorithm 8: A top down alg. for the maximal SEL.

```
   Input: A transition system T = (Init, Tr, Bad)
   Input: An extendable monotone clausal safe trace F of size N
   Output: max(K(F))
 1 i ← N
 2 while i > 0 do
 3 │ if ¬isSat(Tr⟦F^i⟧^{i+1} ∧ Bad(v̄_{N+1})) then break
 4 └ i ← (i − 1)
 5 k ← 1
 6 while k < i + 1 do
 7 │ if ¬isSat(Tr⟦F^i⟧^k ∧ Bad(v̄_{N+1})) then  break
 8 └ k ← (k + 1)
 9 return (i, k)
```

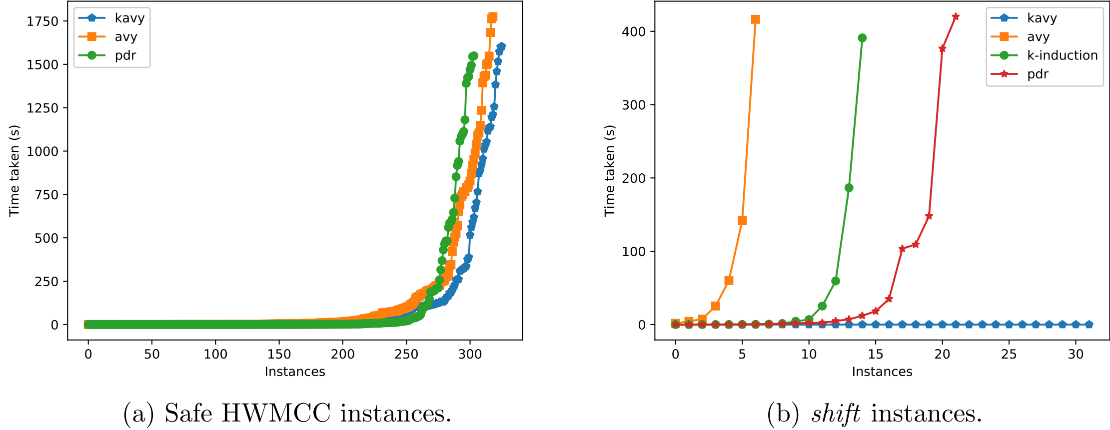
Algorithm 9: A bottom up alg. for the maximal SEL.

Input: A transition system $T = (Init, Tr, Bad)$
Input: An extendable monotone clausal safe trace $\boldsymbol{F}$ of size N
Output: $\max(\mathcal{K}(\boldsymbol{F}))$

```
1  j ← N
2  while j > 0 do
3      if ¬isSat(Tr⟦F^j⟧^1 ∧ Bad(v̄_{N+1})) then break
4      j ← (j − 1)
5  (i, k) ← (j, 1) ; j ← (j + 1) ; ℓ ← 2
6  while ℓ ≤ (j + 1) ∧ j ≤ N do
7      if isSat(Tr⟦F^j⟧^ℓ ∧ Bad(v̄_{N+1})) then ℓ ← (ℓ + 1)
8      else
9          (i, k) ← (j, ℓ)
10         j ← (j + 1)
11 return (i, k)
```

Bottom-up SEL. Alg. 9 searches for a SEL by first finding a maximal regular extension level (line 2) and then searching for larger SELs (lines 6 to 10). Observe that if $(j, \ell) \notin \mathcal{K}(\boldsymbol{F})$, then $\forall p > j \cdot (p, \ell) \notin \mathcal{K}(\boldsymbol{F})$. This is used at line 7 to increase the depth of induction once it is known that $(j, \ell) \notin \mathcal{K}(\boldsymbol{F})$. On the other hand, if $(j, \ell) \in \mathcal{K}(\boldsymbol{F})$, there might be a larger SEL $(j + 1, \ell)$. Thus, whenever a SEL (j, ℓ) is found, it is stored in (i, k) and the search continues (line 10). The algorithm terminates when there are no more valid SEL candidates and returns the last valid SEL. Note that ℓ is incremented only when there does not exists a larger SEL with the current value of ℓ. Thus, for each valid level j, if there exists SELs with level j, the algorithm is guaranteed to find the largest such SEL. Moreover, the level is increased at every possible opportunity. Hence, at the end $(i, k) = \max \mathcal{K}(\boldsymbol{F})$.

In the worst case, Alg. 9 makes at most $3N$ SAT queries. However, compared to Alg. 8, the queries are smaller. Moreover, the computation is incremental and can be aborted with a sub-optimal solution after execution of line 5 or line 9. Note that at line 5, i is a regular extension level (i.e., as in AVY), and every execution of line 9 results in a larger SEL.



(a) Safe HWMCC instances.

(b) *shift* instances.

Figure 4.1: Runtime comparison on SAFE HWMCC instances (a) and *shift* instances (b).

4.3 Evalution

We implemented KAVY on top of the AVY Model Checker[1]. For line 5 of Algorithm 6 we used Algorithm 8. We evaluated KAVY's performance against a version of AVY [77] from the Hardware Model Checking Competition 2017 [14], and the PDR engine of ABC [35]. We have used the benchmarks from HWMCC'14, '15, and '17. Benchmarks that are not solved by any of the solvers are excluded from the presentation. The experiments were conducted on a cluster running Intel E5-2683 V4 CPUs at 2.1 GHz with 8GB RAM limit and 30 minutes time limit.

The results are summarized in Table 4.1. The HWMCC has a wide variety of benchmarks. We aggregate the results based on the competition, and also benchmark origin (based on the name). Some named categories (e.g., *intel*) include benchmarks that have not been included in any competition. The first column in Table 4.1 indicates the category. **Total** is the number of all available benchmarks, ignoring duplicates. That is, if a benchmark appeared in multiple categories, it is counted only once. Numbers in brackets indicate the number of instances that are solved uniquely by the solver. For example, KAVY solves 14 instances in *oc8051* that are not solved by any other solver. The VBS column indicates the *Virtual Best Solver* — the result of running all the three solvers in parallel and stopping as soon as one solver terminates successfully.

Overall, KAVY solves more SAFE instances than both AVY and PDR, while taking less

[1]All code, benchmarks, and results are available at https://arieg.bitbucket.io/avy/

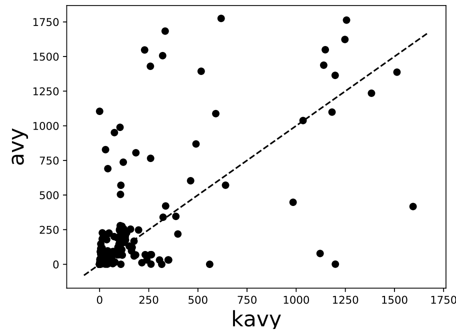
Table 4.1: Summary of instances solved by each tool. Timeouts were ignored when computing the time column.

BENCHMARKS	kAvy			Avy			Pdr			VBS	
	SAFE	UNSAFE	time(m)	SAFE	UNSAFE	time(m)	SAFE	UNSAFE	time(m)	SAFE	UNSAFE
HWMCC' 17	137 (16)	38	499	128 (3)	38	406	109 (6)	40 (5)	174	150	44
HWMCC' 15	193 (4)	84	412	191 (3)	92 (6)	597	194 (16)	67 (12)	310	218	104
HWMCC' 14	49	27 (1)	124	58 (4)	26	258	55 (6)	19 (2)	172	64	29
intel	32 (1)	9	196	32 (1)	9	218	19	5 (1)	40	33	10
6s	73 (2)	20	157	81 (4)	21 (1)	329	67 (3)	14	51	86	21
nusmv	13	0	5	14	0	29	16 (2)	0	38	16	0
bob	30	5	21	30	6 (1)	30	30 (1)	8 (3)	32	31	9
pdt	45	1	54	45 (1)	1	57	47 (3)	1	62	49	1
oski	26	89 (1)	174	28 (2)	92 (4)	217	20	53	63	28	93
beem	10	1	49	10	2	32	20 (8)	7 (5)	133	20	7
oc8051	34 (14)	0	286	20	0	99	6 (1)	1 (1)	77	35	1
power	4	0	25	3	0	3	8 (4)	0	31	8	0
shift	5 (2)	0	1	1	0	18	3	0	1	5	0
necla	5	0	4	7 (1)	0	1	5 (1)	0	4	8	0
prodcell	0	0	0	0	1	28	0	4 (3)	2	0	4
bc57	0	0	0	0	0	0	0	4 (4)	9	0	4
Total	326 (19)	141 (1)	957	319 (8)	148 (6)	1041	304 (25)	117 (17)	567	370	167

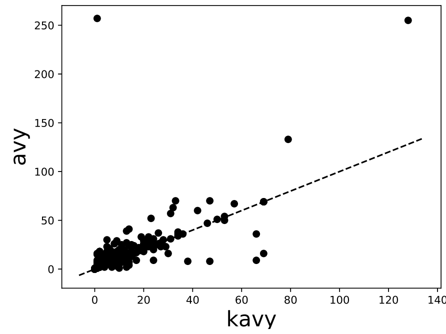
time than Avy (we report time for solved instances, ignoring timeouts). The VBS column shows that kAvy is a promising new strategy, significantly improving overall performance. In the rest of this section, we analyze the results in more detail, provide detailed running-time comparison between the tools, isolate the effect of the new k-inductive strategy and compare the two algorithms that compute SEL.

To compare the running time, we present scatter plots comparing kAvy and Avy (Figure 4.2a), and kAvy and Pdr (Figure 4.2c). In both figures, kAvy is at the bottom. Points above the diagonal are better for kAvy. Compared to Avy, whenever an instance is solved by both solvers, kAvy is often faster, sometimes by orders of magnitude. Compared to Pdr, kAvy and Pdr perform well on very different instances. This is similar to the observation made by the authors of the original paper that presented Avy [77]. Another indicator of performance is the depth of convergence. This is summarized in Figure 4.2b and Figure 4.2d. kAvy often converges much sooner than Avy. The comparison with Pdr is less clear which is consistent with the difference in performance between the two. To get the whole picture, Figure 4.1a presents a cactus plot that compares the running times of the algorithms on all these benchmarks.

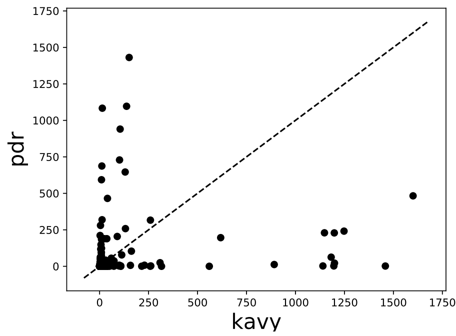
To isolate the effects of k-induction, we compare kAvy to a version of kAvy with k-induction disabled, which we call vanilla. Conceptually, vanilla is similar to Avy since it extends the trace using a 1-inductive extension trace, but its implementation is based on kAvy. The results for the running time and the depth of convergence are shown



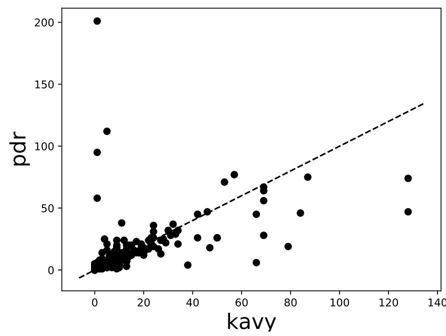
(a) kAvy vs Avy on running time



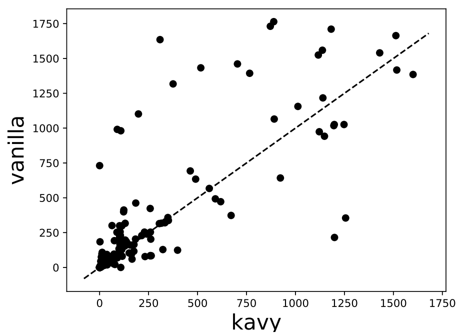
(b) kAvy vs Avy on depth



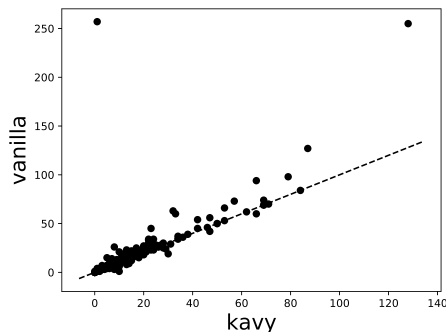
(c) kAvy vs Pdr on running time



(d) kAvy vs Pdr on depth

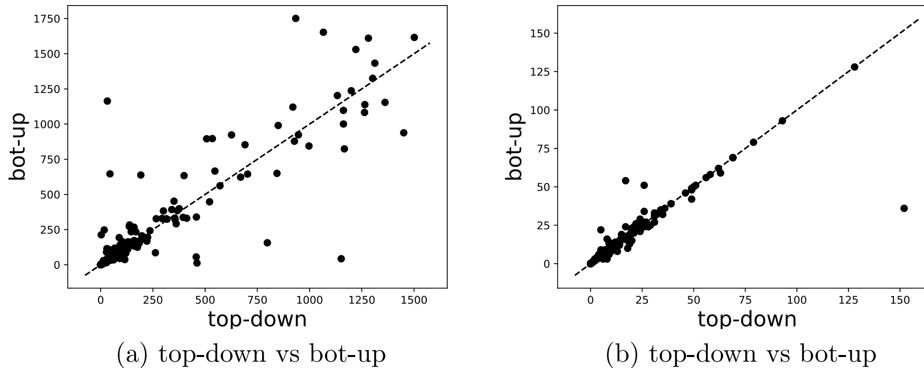


(e) kAvy vs vanilla on running time



(f) kAvy vs vanilla on depth

Figure 4.2: Comparing running time ((a), (c), (e)) and depth of convergence ((b), (d), (f)) of Avy, Pdr and vanilla with kAvy. kAvy is shown on the x-axis. Points above the diagonal are better for kAvy. Only those instances that have been solved by both solvers are shown in each plot.



(a) top-down vs bot-up

(b) top-down vs bot-up

Figure 4.3: Comparing running time (a) and depth of convergence (b) of top-down and bottom-up algorithms

in Figure 4.2e and Figure 4.2f, respectively. The results are very clear — using strong extension traces significantly improves performance and has non-negligible affect on depth of convergence.

We discovered one family of benchmarks, called shift, on which kAvy performs orders of magnitude better than all other techniques. The benchmarks come from encoding bit-vector decision problem into circuits [53, 79]. The shift family corresponds to deciding satisfiability of $(x+y) = (x << 1)$ for two bit-vectors x and y. The family is parameterized by bit-width. The property is k-inductive, where k is the bit-width of x. The results of running Avy, Pdr, k-induction(Algorithm 1)[2], and kAvy are shown in Figure 4.1b. Except for kAvy, all techniques exhibit exponential behavior in the bit-width, while kAvy remains constant. Deeper analysis indicates that kAvy finds a small inductive invariant while exploring just two steps in the execution of the circuit. At the same time, neither inductive generalization nor k-induction alone are able to consistently find the same invariant quickly.

Searching for the maximal strong extension level is a crucial step in kAvy involving many queries to a sat oracle. There are instances where this steps proves to be a bottleneck in kAvy. Figure 4.3 compares kAvy implemented using top-down (Algorithm 8) and bottom-up (Algorithm 9) search for maximal SEL. The top-down search is slightly faster on many instances in the benchmark suite (Figure 4.3a). However, on most instances, kAvy converges at the same depth irrespective of the algorithm used (Figure 4.3b).

[2] We used the k-induction engine `ind` in Abc [22].

Chapter 5

Related work, Conclusions and Future work

Related work. kAvy builds on top of the ideas of Pdr. The use of interpolation for generating an inductive trace is inspired by Avy. While conceptually, our algorithm is similar to Avy, its proof of correctness is non-trivial and is significantly different from that of Avy. We are not aware of any other work that combines interpolation with strong induction.

There are two prior attempts enhancing Pdr-style algorithms with strong induction. Pd-Kind [46] is an SMT-based Model Checking algorithm for infinite-state systems inspired by Pdr. It infers k-inductive invariants driven by the property whereas kAvy infers 1-inductive invariants driven by k-induction. Pd-Kind uses recursive blocking with interpolation and model-based projection to block bad states, and strong induction to propagate (push) lemmas to next level. While the algorithm is very interesting it has not been adapted to the SAT-based setting (i.e. SMC) which makes it impossible to compare on HWMCC instances directly.

The closest related work is KIC3 [41]. It modifies the counterexample queue management strategy in Pdr to utilize strong induction during blocking. The main limitation is that the value for k must be chosen statically ($k = 5$ is reported for the evaluation). kAvy also utilizes strong induction during blocking but computes the value for k dynamically. Unfortunately, the implementation is not available publicly and we could not compare with it directly.

Conclusions. Algorithms that consruct inductive invariants incrementally have displaced those based on strong induction in hardware model checking. In this thesis, we show that strong induction is more concise than induction. We then present kAvy— an SMC algorithm that effectively uses strong induction to guide interpolation and incremental construction of inductive invariants. kAvy searches both for a good inductive strengthening and for the most effective induction depth. We have implemented kAvy on top of the Avy Model Checker. The experimental results on HWMCC instances show that our approach is effective.

Future work. In [46], it was suggested that proving a separation between strong induction and induction would entail new complexity results on quantifier elimination. This might be a fruitful direction to extend the theory results from this thesis.

The inductive invariants generated by both Avy and kAvy depend on the interpolant generated by the underlying SAT solver. One way to guide the interpolant is to abstract away parts of the transision relation before generating the interpolant. This was shown to be effective in Avy. Preliminary experiments on kAvy also showed that under the right abstraction, kAvy could converge at a lower depth on many benchmarks. However, the overhead of finding a good abstraction outwieghed the benefits. Reducing this overhead is a challenge.

One of the current limitations of kAvy is that it is not compatible with aggressive simplifications during BMC [78]. This was a big let down especially in the $6s$ class of benchmarks. Addressing this is left for future work.

The search for the maximal SEL is an overhead in kAvy. There could be benchmarks in which this overhead outweighs its benefits. However, we have not come across such benchmarks so far. In such cases, kAvy can choose to settle for a sub-optimal SEL as mentioned in section 4.2. Deciding when and how much to settle for remains a challenge.

References

[1] *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2000, October 30 - Novemver 5, 2000, Takamatsu, Japan*. IEEE, 2000.

[2] *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 2009.

[3] *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013.

[4] Erika Ábrahám and Klaus Havelund, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*. Springer, 2014.

[5] Rajeev Alur and Doron A. Peled, editors. *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*. Springer, 2004.

[6] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. An adaptive parallel SAT solver. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 30–48, 2016.

[7] Julia M. Badger and Kristin Yvonne Rozier, editors. *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*. Springer, 2019.

[8] Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors. *Fields of Logic and Computation II - Essays Dedicated to Yuri*

Gurevich on the Occasion of His 75th Birthday, volume 9300 of *Lecture Notes in Computer Science*. Springer, 2015.

[9] Anton Belov and João Marques-Silva. MUSer2: An Efficient MUS Extractor. *JSAT*, 8(3/4):123–128, 2012.

[10] Ryan Berryhill, Alexander Ivrii, Neil Veira, and Andreas G. Veneris. Learning support sets in IC3 and Quip: The good, the bad, and the ugly. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 140–147, 2017.

[11] Olaf Beyersdorff and Christoph M. Wintersteiger, editors. *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*, volume 10929 of *Lecture Notes in Computer Science*. Springer, 2018.

[12] Armin Biere and Roderick Bloem, editors. *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*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.

[13] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, pages 193–207, 1999.

[14] Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking competition 2017. In Stewart and Weissenbacher [74], page 9.

[15] Per Bjesse and Anna Slobodová, editors. *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*. FMCAD Inc., 2011.

[16] Nikolaj Bjørner and Arie Gurfinkel, editors. *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. IEEE, 2018.

[17] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II -*

Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday, pages 24–51, 2015.

[18] Roderick Bloem and Natasha Sharygina, editors. *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. IEEE, 2010.

[19] Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors. *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*. Springer, 2011.

[20] 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*, pages 70–87, 2011.

[21] Guillaume Brat, Neha Rungta, and Arnaud Venet, editors. *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*. Springer, 2013.

[22] Robert K. Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *CAV*, pages 24–40, 2010.

[23] Ed Brinksma and Kim Guldstrand Larsen, editors. *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.

[24] Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors. *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000*, volume 2067 of *Lecture Notes in Computer Science*. Springer, 2001.

[25] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind 2 Model Checker. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 510–517, 2016.

[26] Swarat Chaudhuri and Azadeh Farzan, editors. *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*. Springer, 2016.

[27] Alessandro Cimatti. Industrial applications of model checking. In Cassez et al. [24], pages 153–168.

[28] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In Ábrahám and Havelund [4], pages 46–61.

[29] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

[30] Rance Cleaveland, editor. *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*. Springer, 1999.

[31] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[32] Nadia Creignou and Daniel Le Berre, editors. *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*. Springer, 2016.

[33] Leonardo Mendonça de Moura, Sam Owre, Harald Rueß, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 496–500, 2004.

[34] Niklas Eén, Alan Mishchenko, and Nina Amla. A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 181–188, 2010.

[35] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134, 2011.

[36] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[37] Chuchu Fan, Bolun Qi, Sayan Mitra, and Mahesh Viswanathan. Dryvr: Data-driven verification and compositional reasoning for automotive systems. In Majumdar and Kuncak [59], pages 441–461.

[38] Pierre-Loïc Garoche, Temesghen Kahsai, and Cesare Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, pages 139–154, 2013.

[39] Bernhard Gramlich, Dale Miller, and Uli Sattler, editors. *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*. Springer, 2012.

[40] Arie Gurfinkel and Alexander Ivrii. Pushing to the top. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 65–72, 2015.

[41] Arie Gurfinkel and Alexander Ivrii. K-induction without unrolling. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 148–155, 2017.

[42] Hamed Hatami, Pierre McKenzie, and Valerie King, editors. *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*. ACM, 2017.

[43] 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*, pages 181–188, 2013.

[44] 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*, pages 355–370, 2012.

[45] Ranjit Jhala and David A. Schmidt, editors. *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*. Springer, 2011.

[46] Dejan Jovanovic and Bruno Dutertre. Property-directed k-induction. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 85–92, 2016.

[47] Warren A. Hunt Jr. and Steven D. Johnson, editors. *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*. Springer, 2000.

[48] Warren A. Hunt Jr. and Fabio Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003.

[49] Hari Govind V. K., Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. Interpolating strong induction. *CoRR*, abs/1906.01583, 2019.

[50] Temesghen Kahsai, Yeting Ge, and Cesare Tinelli. Instantiation-based invariant discovery. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 192–206, 2011.

[51] Roope Kaivola and Thomas Wahl, editors. *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*. IEEE, 2015.

[52] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[53] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.*, 59(2):323–376, 2016.

[54] Oliver Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*. Springer, 2009.

[55] Robert P. Kurshan. Transfer of model checking to industrial practice. In Clarke et al. [29], pages 763–793.

[56] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 123–140, 2016.

[57] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for CDCL SAT solvers. 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*, pages 94–110, 2018.

[58] Bjørnar Luteberget, Koen Claessen, and Christian Johansen. Design-time railway capacity verification using SAT modulo discrete event simulation. In Bjørner and Gurfinkel [16], pages 1–9.

[59] Rupak Majumdar and Viktor Kuncak, editors. *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*. Springer, 2017.

[60] Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 250–264, 2002.

[61] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 1–13, 2003.

[62] Kenneth L. McMillan. Interpolation and model checking. In *Handbook of Model Checking.*, pages 421–446. 2018.

[63] Alain Mebsout and Cesare Tinelli. Proof certificates for SMT-based model checkers for infinite-state systems. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 117–124, 2016.

[64] Alan Mishchenko, Michael L. Case, Robert K. Brayton, and Stephen Jang. Scalable and scalably-verifiable sequential synthesis. In *2008 International Conference on Computer-Aided Design, ICCAD 2008, San Jose, CA, USA, November 10-13, 2008*, pages 234–241, 2008.

[65] Sani R. Nassif and Jaijeet S. Roychowdhury, editors. *2008 International Conference on Computer-Aided Design, ICCAD 2008, San Jose, CA, USA, November 10-13, 2008*. IEEE Computer Society, 2008.

[66] Antti Pakonen, Topi Tahvonen, Markus Hartikainen, and Mikko Pihlanko. Practical applications of model checking in the finnish nuclear industry. In *10th International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, NPIC and HMIT 2017*, volume 2, pages 1342–1352, United States, 1 2017. American Nuclear Society ANS. Project: 113347.

[67] Ruzica Piskac and Muralidhar Talupur, editors. *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. IEEE, 2016.

[68] Toniann Pitassi and Robert Robere. Strongly exponential lower bounds for monotone computation. In Hatami et al. [42], pages 1246–1255.

[69] Michel Rueher, editor. *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*. Springer, 2016.

[70] Claude E Shannon. The synthesis of two-terminal switching circuits. In *The Bell System Technical Journal*, pages 59–98, 1949.

[71] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, pages 108–125, 2000.

[72] Reid G. Simmons, Charles Pecheur, and Grama Srinivasan. Towards automatic verification of autonomous systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2000, October 30 - Novemver 5, 2000, Takamatsu, Japan* [1], pages 1410–1415.

[73] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Kullmann [54], pages 244–257.

[74] Daryl Stewart and Georg Weissenbacher, editors. *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. IEEE, 2017.

[75] Tayssir Touili, Byron Cook, and Paul Jackson, editors. *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*. Springer, 2010.

[76] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 1–8, 2009.

[77] Yakir Vizel and Arie Gurfinkel. Interpolating property directed reachability. 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*, pages 260–276, 2014.

[78] Yakir Vizel, Arie Gurfinkel, and Sharad Malik. Fast Interpolating BMC. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 641–657, 2015.

[79] Yakir Vizel, Alexander Nadel, and Sharad Malik. Solving linear arithmetic with SAT-based model checking. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 47–54, 2017.