Interpolating Strong Induction

Hari Govind V K[1], Yakir Vizel[2], Vijay Ganesh[1], and Arie Gurfinkel[1]

[1] University of Waterloo
[2] The Technion

Abstract. The principle of strong induction, also known as k-induction is one of the first techniques for unbounded SAT-based Model Checking (SMC). While elegant and simple to apply, properties as such are rarely k-inductive 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 paper, we present kAvy, an SMC algorithm that effectively uses k-induction to guide interpolation and Pdr-style inductive generalization. Unlike pure k-induction, kAvy uses Pdr-style generalization to compute and strengthen an inductive trace. Unlike pure Pdr, kAvy uses relative k-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 k-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 k-induction.

1 Introduction

The principle of strong induction, also known as k-induction, is a generalization of (simple) induction that extends the base- and inductive-cases to k steps of a transition system [27]. A safety property P is k-inductive in a transition system T iff (a) P is true in the first $(k-1)$ steps of T, and (b) if P is assumed to hold for $(k-1)$ consecutive steps, then P holds in k steps of T. Simple induction is equivalent to 1-induction. Unlike induction, strong induction is complete for safety properties: a property P is safe in a transition system T iff there exists a natural number k such that P is k-inductive in T (assuming the usual restriction to simple paths). This makes k-induction a powerful method for unbounded SAT-based Model Checking (SMC).

Unlike other SMC techniques, strong induction reduces model checking to pure SAT that does not require any additional features such as solving with assumptions [12], interpolation [24], resolution proofs [17], Maximal Unsatisfiable Subsets (MUS) [2], etc. It easily integrates with existing SAT-solvers and immediately benefits from any improvements in heuristics [23,22], pre- and inprocessing [18], and parallel solving [1]. The simplicity of applying k-induction made it the go-to technique for SMT-based infinite-state model checking [9,11,19]. In that context, it is particularly effective in combination with invariant synthesis [20,14]. Moreover, for some theories, strong induction is strictly stronger than 1-induction [19]: there are properties that are k-inductive, but have no 1-inductive strengthening.

Notwithstanding all of its advantages, strong induction has been mostly displaced by more recent SMC techniques such as Interpolation [25], Property Directed Reachability [7,13,15,3], and their combinations [29]. In SMC k-induction is equivalent to induction: any k-inductive property P can be strengthened to an inductive property Q [16,6]. Even though in the worst case Q is exponentially larger than P [6], this is rarely observed in practice [26]. Furthermore, the SAT queries get very hard as k increases and usually succeed only for rather small values of k. A recent work [16] shows that strong induction can be integrated in Pdr. However, [16] argues that k-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 [16], k is fixed to 5, and regular induction is used as soon as 5-induction fails.

In this paper, we present kAvy, an SMC algorithm that effectively uses k-induction to guide interpolation and Pdr-style inductive generalization. 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 k-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 k-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, k-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 a system in Fig. 1 that counts upto 64 and resets. The property, $p : c < 66$, is 2-inductive. IC3, 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.

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

Fig. 1. An example system.

kAvy builds upon the foundations of Avy [29]. Avy first uses Bounded Model Checking [4] (BMC) to prove that the property P holds up to bound N. Then, it uses a sequence interpolant [28] and Pdr-style inductive-generalization [7] to construct 1-inductive strengthening candidate for P. We emphasize that using k-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 [7].

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 either of Pdr and Avy. For a specific family of examples from [21], kAvy exhibits nearly constant time performance, compared to an exponential growth of Avy, Pdr, and k-induction (see Fig. 2b in Section 5). This further emphasizes the effectiveness of efficiently integrating strong induction into modern SMC.

The rest of the paper is structured as follows. After describing the most relevant related work, we present the necessary background in Section 2 and give an overview of SAT-based model checking algorithms in Section 3. kAvy is presented in Section 4, followed by presentation of results in Section 5. Finally, we conclude the paper in Section 6.

Related work. kAvy builds on top of the ideas of IC3 [7] and Pdr [13]. The use of interpolation for generating an inductive trace is inspired by Avy [29]. 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 k-induction. Pd-Kind [19] is an SMT-based Model Checking algorithm for infinite-state systems inspired by IC3/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 k-induction to propagate (push) lemmas to next level. While the algorithm is very interesting it is hard to adapt it to SAT-based setting (i.e. SMC), and impossible to compare on HWMCC instances directly.

The closest related work is KIC3 [16]. It modifies the counter example queue management strategy in IC3 to utilize k-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 k-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.

2 Background

In this section, we present notations and background that is required for the description of our algorithm.

Safety Verification. A symbolic 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 $\mathit{Init} \Rightarrow P$.

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:

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

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 $s \in \mathit{Bad}$ 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:

$$\mathit{Init}(\bar{v}_0) \wedge \mathit{Tr}^N \wedge \mathit{Bad}(\bar{v}_N) \tag{2}$$

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

An *inductive invariant* is a formula Inv that satisfies:

$$\mathit{Init}(\bar{v}) \Rightarrow \mathit{Inv}(\bar{v}) \qquad\qquad \mathit{Inv}(\bar{v}) \wedge \mathit{Tr}(\bar{v}, \bar{v}') \Rightarrow \mathit{Inv}(\bar{v}') \tag{3}$$

A transition system T is SAFE iff there exists an inductive invariant Inv s.t. $\mathit{Inv}(\bar{v}) \Rightarrow P(\bar{v})$. In this case we say that Inv is a *safe* inductive invariant.

The *safety* verification problem is to decide whether a transition system T is SAFE or UNSAFE, i.e., whether there exists a safe inductive invariant or a counterexample.

Strong Induction. Strong induction (or k-induction) is a generalization of the notion of an inductive invariant that is similar to how "simple" induction is generalized in mathematics. 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: $\mathit{Init}(\bar{v}_0) \wedge \mathit{Tr}^k \Rightarrow \left(\bigwedge_{i=0}^{k} \mathit{Inv}(\bar{v}_i)\right)$. 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: $\mathit{Tr}[\mathit{Inv}]^k \Rightarrow \mathit{Inv}(\bar{v}_k)$. 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 $\mathit{Inv} \Rightarrow P$, 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 inductive invariant w.r.t. T iff there exists a safe k-inductive invariant w.r.t. T.*

Theorem 1 states that k-induction principle is as complete as 1-induction. One direction is trivial (since we can take $k = 1$). The other 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 [6]. In practice, both invariants tend to be of similar size.

We say that a formula φ is *k-inductive relative* to F if it is a $(k-1)$-invariant and $Tr[\varphi \wedge F]^k \Rightarrow \varphi(\bar{v}_k)$.

Craig Interpolation [10]. We use an extension of Craig Interpolants to sequences, which is common in Model Checking. Let $\boldsymbol{A} = [A_1, \ldots, A_N]$ 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 $\boldsymbol{A}$.

3 SAT-based Model Checking

In this section, we give a brief overview of SAT-based Model Checking algorithms: IC3/Pdr [7,13], and Avy [29]. While these algorithms are well-known, we give a uniform presentation and establish notation necessary for the rest of the paper. We fix a symbolic transition system $T = (\bar{v}, Init, Tr, Bad)$.

The main data-structure of these algorithms 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:

$$Init(\bar{v}) = F_0(\bar{v}) \qquad \forall 0 \leq i < N \cdot F_i(\bar{v}) \wedge Tr(\bar{v}, \bar{v}') \Rightarrow F_{i+1}(\bar{v}') \tag{4}$$

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 $\exists 1 \leq i \leq N \cdot F_i \Rightarrow \left(\bigvee_{j=0}^{i-1} F_j\right)$.

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

$$Tr[\boldsymbol{F}^k] = \bigwedge_{i=k}^{|F|} F_i(\bar{v}_i) \wedge Tr(\bar{v}_i, \bar{v}_{i+1}) \tag{5}$$

We write $Tr[\boldsymbol{F}]$ to denote an unrolling of a 0-suffix of $\boldsymbol{F}$ (i.e $\boldsymbol{F}$ itself). 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 admit counterexamples of length less than N. 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. SAT-based model checking algorithms 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 $\mathcal{W}(\boldsymbol{F})$. The existence of an extension level i implies that an unrolling of the i-suffix does not contain any *Bad* states:

Proposition 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.*

Example 1. For Fig. 1, $\boldsymbol{F} = [c = 0, c < 66]$ is a safe trace of size 1. The formula $(c < 66) \wedge Tr \wedge \neg(c' < 66)$ is satisfiable. Therefore, there does not exists an extension trace at level 1. Since $(c = 0) \wedge Tr \wedge (c' < 66) \wedge Tr' \wedge (c'' \geq 66)$ 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 Avy and Pdr through the lens of extension level. The goal of this presentation is to make the paper self-contained. We omit many important optimization details, and refer the reader to the original papers [13,7,29].

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 can reach *Bad* (called *bad states*). A bad state is blocked at the largest level possible. Alg. 1 shows PdrBlock, the backward search procedure that identifies and blocks bad states. 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 satistifies 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. PdrIndGen is a crucial optimization to Pdr. However, we do not explain it for the sake of simplicity. 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 if the clauses it learnt are also true at higher levels. Pdr terminates when the trace is closed.

Avy, shown in Alg. 2, 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, a sequence interpolant is extracted from the unsatisfiability of $Tr[\boldsymbol{F}^{\max(\mathcal{W})}] \wedge Bad(\bar{v}_{N+1})$. A longer trace

Algorithm 1: 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  $w \leftarrow N + 1$ ; $\boldsymbol{G} \leftarrow \boldsymbol{F}$ ; $Q.push(\langle Bad, N+1 \rangle)$
2  while $\neg Q.empty()$ do
3      $\langle s, d \rangle \leftarrow Q.pop()$
4      if $d == 0$ then return [ ]
5      if isSat($F_{d-1}(\bar{v}) \wedge Tr(\bar{v}, \bar{v}') \wedge s(\bar{v}')$) then
6          $t \leftarrow predecessor(s)$
7          $Q.push(t, d-1)$
8          $Q.push(s, d)$
9      else
10         $\forall 0 \leq i \leq d \cdot G_i \leftarrow$
             $(G_i \wedge$ PdrIndGen$(\neg s))$
11         $w \leftarrow min(w, d)$
12 return $\boldsymbol{G}$
```

Algorithm 2: Avy.

```
   Input: A transition system
          $T = (Init, Tr, Bad)$
   Output: safe/unsafe
1  $F_0 \leftarrow Init$ ; $N \leftarrow 0$
2  repeat
3      if isSat($Tr[\boldsymbol{F}^0] \wedge Bad(\bar{v}_{N+1})$) then
         return unsafe
4      $k \leftarrow \max\{i \mid \neg$isSat$(Tr[\boldsymbol{F}^i] \wedge Bad(\bar{v}_{N+1}))\}$
5      $I_{k+1}, \ldots, I_{N+1} \leftarrow$
         seqItp($Tr[\boldsymbol{F}^k] \wedge Bad(\bar{v}_{N+1})$)
6      $\forall 0 \leq i \leq k \cdot G_i \leftarrow F_i$
7      $\forall k < i \leq (N+1) \cdot G_i \leftarrow F_i \wedge I_i$
8      $\boldsymbol{F} \leftarrow$ AvyMkTrace$([G_0, \ldots, G_{N+1}])$
9      $\boldsymbol{F} \leftarrow$ PdrPush$(\boldsymbol{F})$
10     if $\exists 1 \leq i \leq N \cdot F_i \Rightarrow \left(\bigvee_{j=0}^{i-1} F_j\right)$ then
         return safe
11     $N \leftarrow N + 1$
12 until $\infty$
```

$\boldsymbol{G} = [G_0, \ldots, G_N, G_{N+1}]$ is constructed using the sequence interpolant (line 7). Observe that $\boldsymbol{G}$ is an extension trace of $\boldsymbol{F}$. While $\boldsymbol{G}$ is safe, it is neither monotone nor clausal. A helper routine AvyMkTrace is used to convert $\boldsymbol{G}$ to a proper Pdr trace on line 8 (see [29] for the details on AvyMkTrace). Avy converges when the trace is closed.

4 Interpolating k-Induction

In this section, we present kAvy, an SMC algorithm that uses the principle of strong induction to extend an inductive trace. The section is structured as follows. First, we introduce a concept of extending a trace using relative k-induction. Second, we present kAvy and describe the details of how k-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.

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, 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

Algorithm 3: 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 $\infty$
```

denote this *characteristic formula* for SEL (i, k) :

$$Tr[\![\boldsymbol{F}^i]\!]^k = \begin{cases} Tr[F_i]_{i+1-k}^{i+1} \wedge Tr[\boldsymbol{F}^{i+1}] & \text{if } 0 \leq i < N \\ Tr[F_N]_{N+1-k}^{N+1} & \text{if } i = N \end{cases} \tag{6}$$

Proposition 2. *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.*

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 1. *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, 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. 3. 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). 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 Alg. 4) 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 Proposition 2, $\Psi = Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1})$ is unsatisfiable. Let $\mathcal{A} = \{A_{i-k+1}, \ldots, A_{N+1}\}$ 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:

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

Note that in $(\heartsuit)$, 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 2. *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, $\forall i-k+2 \leq j \leq i \cdot Tr[F_i]^j_{i-k+2} \Rightarrow F_{i+1}(\bar{v}_{j+1})$. Moreover, by $(\heartsuit)$, $F_i \wedge Tr \Rightarrow I'_{i-k+2}$ and $\forall i-k+2 \leq j \leq i+1 \cdot (F_i \wedge I_j) \wedge Tr \Rightarrow I'_{j+1}$. Equivalently, $\forall i-k+2 \leq j \leq i+1 \cdot Tr[F_i]^j_{i-k+2} \Rightarrow I_{j+1}(\bar{v}_{j+1})$. 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 2 to define a strong extension trace $\boldsymbol{G}$:

Lemma 3. *Let $\boldsymbol{G} = [G_0, \ldots, G_{N+1}]$, be an inductive trace defined as follows:*

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

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

Algorithm 4: 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 $N+1$
$I_{i-k+2}, \ldots, I_{N+1} \leftarrow$ seqItp$(Tr[\![\boldsymbol{F}^i]\!]^k \wedge Bad(\bar{v}_{N+1}))$
$\boldsymbol{G} \leftarrow [F_0, \ldots, F_N, \top]$
for $j \leftarrow i-k+1$ to $i$ do
    $P_j \leftarrow (G_j \vee (G_{i+1} \wedge I_{j+1}))$
    // Inv$_1$: $\boldsymbol{G}$ is monotone and clausal
    // Inv$_2$: $G_i \wedge Tr \Rightarrow P_j$
    // Inv$_3^\dagger$ : $\forall j < m \leq (i+1) \cdot G_m \equiv F_m \wedge \bigwedge_{\ell=i-k+1}^{j-1} (G_\ell \vee I_{\ell+1})$
    // Inv$_3$ : $\forall j < m \leq (i+1) \cdot G_m \Rightarrow F_m \wedge \bigwedge_{\ell=i-k+1}^{j-1} (G_\ell \vee I_{\ell+1})$
    $[\_, \_, G_{i+1}] \leftarrow$ PdrBlock$([Init, G_i, G_{i+1}], (Init, Tr, \neg P_j))$
$P_i \leftarrow (G_i \vee (G_{i+1} \wedge I_{j+1}))$
if $i = 0$ then $[\_, \_, G_{i+1}] \leftarrow$ PdrBlock$([Init, G_{i+1}], (Init, Tr, \neg P_i))$
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$_4$: $G_{i+1} \Rightarrow F_{i+1} \wedge \bigwedge_{\ell=i-k+1}^{i} (G_\ell \vee I_{\ell+1})$
for $j \leftarrow i+1$ to $N+1$ do
    $P_j \leftarrow G_j \vee (G_{j+1} \wedge I_{j+1})$
    // Inv$_6$: $G_j \wedge Tr \Rightarrow P_j$
    $[\_, \_, G_{j+1}] \leftarrow$ PdrBlock$([Init, G_j, G_{j+1}], (Init, Tr, \neg P_j))$
    $\boldsymbol{G} \leftarrow$ 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 $\boldsymbol{F}$
return $\boldsymbol{G}$
```

Proof. By Lemma 2, 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, $\forall 0 \leq j \leq N \cdot G_j \Rightarrow F_j$. By (♡), $F_i \wedge Tr \Rightarrow I'_{i-k+2}$ and $\forall i-k+2 \leq j < i+2 \cdot (F_i \wedge I_j) \wedge Tr \Rightarrow I'_{j+1}$. 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 $i-k+2 \leq j < i+2$. Since $\boldsymbol{F}$ is monotone, $\forall i-k+2 \leq j < i+2 \cdot \left((F_j \wedge \bigwedge_{m=i-k+2}^{j} I_m) \wedge Tr\right) \Rightarrow \bigwedge_{m=i-k+2}^{j+1} I'_m$

By (♡), $\forall i < j \leq N \cdot (F_j \wedge I_j) \wedge Tr \Rightarrow I'_{j+1}$. 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 $0 \leq j \leq N$. Since $\boldsymbol{F}$ is safe and $I_{N+1} \Rightarrow \neg Bad$, then $\boldsymbol{G}$ is safe and stronger than $\boldsymbol{F}$. □

Lemma 3 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 3), but uses PdrBlock to systematically construct a safe monotone clausal extension of $\boldsymbol{F}$.

The procedure kAvyExtend is shown in Alg. 4. 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. 4. 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 Alg 4, 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 2, 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 $\mathtt{Inv}_2$. It holds on loop entry since $G_i \wedge Tr \Rightarrow I_{i-k+2}$ (since $G_i = F_i$ and ($\heartsuit$)) and $G_i \wedge Tr \Rightarrow G_{i+1}$ (since $\boldsymbol{G}$ is initially a trace). Let G_i and G_i^* be the i^{th} 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 $\mathtt{Inv}_2$ 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, $\mathtt{Inv}_2$ holds at the end of the iteration.

$\mathtt{Inv}_2$ 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 $\mathtt{Inv}_2$, 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 j^{th} iteration. Because of this, the property P_j in this phase is slightly different than that of Phase 1. Correctness follows from invariant $\mathtt{Inv}_6$ 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 2. 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 2. *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. 3. 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

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

Algorithm 5: A top down 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 $i \leftarrow N$
2 while $i > 0$ do
3     if $\neg$isSat( $Tr[\![\boldsymbol{F}^i]\!]^{i+1} \wedge Bad(\bar{v}_{N+1})$)
        then break
4     $i \leftarrow (i - 1)$
5 $k \leftarrow 1$
6 while $k < i + 1$ do
7     if $\neg$isSat( $Tr[\![\boldsymbol{F}^i]\!]^{k} \wedge Bad(\bar{v}_{N+1})$)
        then break
8     $k \leftarrow (k + 1)$
9 return $(i, k)$
```

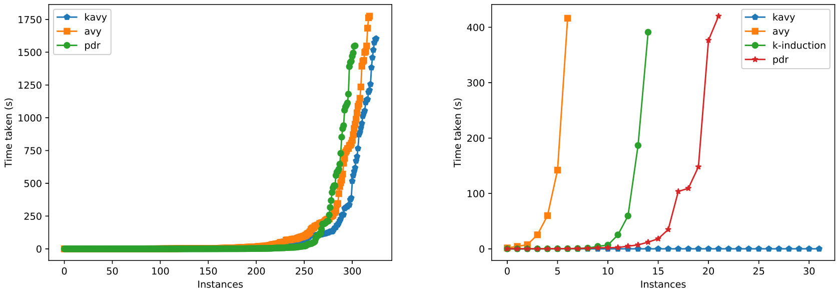
Algorithm 6: A bottom up alg. for the maximal SEL.

```
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  $j \leftarrow N$
2  while $j > 0$ do
3      if $\neg$isSat( $Tr[\![\boldsymbol{F}^j]\!]^{1} \wedge Bad(\bar{v}_{N+1})$)
         then break
4      $j \leftarrow (j - 1)$
5  $(i, k) \leftarrow (j, 1)$ ; $j \leftarrow (j + 1)$ ; $\ell \leftarrow 2$
6  while $\ell \leq (j + 1) \wedge j \leq N$ do
7      if isSat( $Tr[\![\boldsymbol{F}^j]\!]^{\ell} \wedge Bad(\bar{v}_{N+1})$)
         then $\ell \leftarrow (\ell + 1)$
8      else
9          $(i, k) \leftarrow (j, \ell)$
10         $j \leftarrow (j + 1)$
11 return $(i, k)$
```

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. 5. 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.

Bottom-up SEL. Alg. 6 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. 6 makes at most $3N$ SAT queries. However, compared to Alg. 5, 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.

Fig. 2. Runtime comparison on SAFE HWMCC instances (a) and *shift* instances (b).

5 Evaluation

We implemented kAvy on top of the Avy Model Checker[3]. For line 5 of Alg. 3 we used Alg 5. We evaluated kAvy's performance against a version of Avy [29] from the Hardware Model Checking Competition 2017 [5], and the Pdr engine of ABC [13]. 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 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 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 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 run-time comparison between the tools, and isolate the effect of the new k-inductive strategy.

To compare the running time, we present scatter plots comparing kAvy and Avy (Fig. 3a), and kAvy and Pdr (Fig. 3b). In both figures, kAvy is at the

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

Table 1. Summary of instances solved by each tool. Timeouts were ignored when computing the time column.

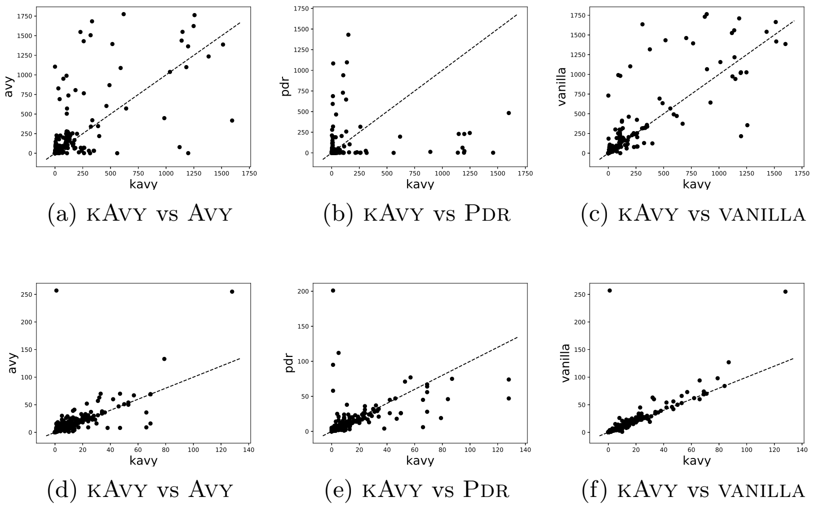
BENCHMARKS	kAvy			Avy			Pdr			VBS	
	SAFE	UNSAFE	time(m)	SAFE	UNSAFE	time(m)	SAFE	UNSAFE	time(m)	SAFE	UNSAFE
HWMCC' 17	137 (16)	38	499	128 (3)	38	406	109 (6)	40 (5)	174	150	44
HWMCC' 15	193 (4)	84	412	191 (3)	92 (6)	597	194 (16)	67 (12)	310	218	104
HWMCC' 14	49	27 (1)	124	58 (4)	26	258	55 (6)	19 (2)	172	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

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 [29]. Another indicator of performance is the depth of convergence. This is summarized in Fig. 3d and Fig. 3e. 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, Fig. 2a 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 in Fig. 3c and Fig. 3f, respectively. The results are very clear — using strong extension traces significantly improves performance and has non-negligible affect on depth of convergence.

Finally, 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 [21,30]. The shift family corresponds to deciding satisfiability of $(x + y) = (x << 1)$ for two bit-vecors 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[4], and kAvy are shown in Fig. 2b. 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

[4] We used the k-induction engine `ind` in Abc [8].



(a) kAvy vs Avy (b) kAvy vs Pdr (c) kAvy vs vanilla

(d) kAvy vs Avy (e) kAvy vs Pdr (f) kAvy vs vanilla

Fig. 3. Comparing running time ((a), (b), (c)) and depth of convergence ((d), (e), (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.

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.

6 Conclusion

In this paper, we present kAvy— an SMC algorithm that effectively uses k-inductive reasoning to guide interpolation and inductive generalization. kAvy searches both for a good inductive strengthening and for the most effective induction depth k. We have implemented kAvy on top of Avy Model Checker. The experimental results on HWMCC instances show that our approach is effective.

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.

Acknowledgements We thank the anonymous reviewers and Oded Padon for their thorough review and insightful comments. This research was enabled in part by support provided by Compute Ontario (`https://computeontario.ca/`), Compute Canada (`https://www.computecanada.ca/`) and the grants from Natural Sciences and Enginerring Research Council Canada.

References

1. 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.
2. Anton Belov and João Marques-Silva. MUSer2: An Efficient MUS Extractor. *JSAT*, 8(3/4):123–128, 2012.
3. 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.
4. 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, E-TAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, pages 193–207, 1999.
5. Armin Biere, Tom van Dijk, and Keijo Heljanko. Hardware model checking competition 2017. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, page 9. IEEE, 2017.
6. 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.
7. 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.
8. Robert K. Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *CAV*, pages 24–40, 2010.
9. 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.
10. William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.
11. 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.
12. 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.
13. 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.

14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. *Theory Comput. Syst.*, 59(2):323–376, 2016.
22. 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.
23. 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.
24. 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.
25. Kenneth L. McMillan. Interpolation and model checking. In *Handbook of Model Checking.*, pages 421–446. 2018.
26. 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.
27. 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.
28. 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.

29. 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.
30. 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.