Interpolating Strong Induction

Hari Govind Vediramana Krishnan\textsuperscript{1(✉)}, Yakir Vizel\textsuperscript{2}, Vijay Ganesh\textsuperscript{1}, and Arie Gurfinkel\textsuperscript{1}

\textsuperscript{1} University of Waterloo, Waterloo, Canada
\texttt{hgvedira@uwaterloo.ca}
\textsuperscript{2} The Technion, Haifa, Israel

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 K\textsc{Avy}, an SMC algorithm that effectively uses \(k\)-induction to guide interpolation and PDr-style inductive generalization. Unlike pure \(k\)-induction, K\textsc{Avy} uses PDr-style generalization to compute and strengthen an inductive trace. Unlike pure PDr, K\textsc{Avy} 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 K\textsc{Avy} within the Avy Model Checker and evaluated it on HWMCC instances. Our results show that K\textsc{Avy} 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 \textit{shift}, K\textsc{Avy} 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 [22,23], pre-and in-processing [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 [14,20]. 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 [3,7,13,15], and their combinations [29]. In SMC $k$-induction is equivalent to induction: any $k$-inductive property $P$ can be strengthened to an inductive property $Q$ [6,16]. 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 up to 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 $(c \neq 65) \land (c < 66)$, and terminates.

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 Sect. 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 Sect. 2 and give an overview of SAT-based model checking algorithms in Sect. 3. KAVY is presented in Sect. 4, followed by presentation of results in Sect. 5. Finally, we conclude the paper in Sect. 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 $(\bar{v}, \text{Init}, \text{Tr}, \text{Bad})$, 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 $\bar{v}' = \{v' | v \in \bar{v}\}$ 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 $\text{Tr}$ 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 $\phi$, we use $\phi(\bar{v}')$, or $\phi'$ 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 $P = \neg \text{Bad}$ is true in the initial state, that is $\text{Init} \Rightarrow P$.

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

$$\text{Tr}[^{N} \phi]_{M} = \bigwedge_{i=M}^{N-1} \phi(\bar{v}_{i}) \land \text{Tr}(\bar{v}_{i}, \bar{v}_{i+1}) \quad (1)$$

We write $\text{Tr}[^{N} \phi]$ when $M = 0$ and $\text{Tr}[^{N} \phi]$ when $\phi = \top$.

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

$$\text{Init}(\bar{v}_{0}) \land \text{Tr}[^{N}] \land \text{Bad}(\bar{v}_{N}) \quad (2)$$

$T$ is SAFE if no such $N$ exists. Whenever $T$ is UNSAFE and $s_{N} \in \text{Bad}$ is a reachable state, the path from $s_{0} \in \text{Init}$ to $s_{N}$ is called a *counterexample*.

An inductive invariant is a formula $\text{Inv}$ that satisfies:

$$\text{Init}(\bar{v}) \Rightarrow \text{Inv}(\bar{v}) \quad \text{Inv}(\bar{v}) \land \text{Tr}(\bar{v}, \bar{v}') \Rightarrow \text{Inv}(\bar{v}') \quad (3)$$

A transition system $T$ is SAFE iff there exists an inductive invariant $\text{Inv}$ s.t. $\text{Inv}(\bar{v}) \Rightarrow P(\bar{v})$. In this case we say that $\text{Inv}$ 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 $\text{Inv}$ 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: $\text{Init}(\bar{v}_{0}) \land \text{Tr}^{k} \Rightarrow \left(\bigwedge_{i=0}^{k} \text{Inv}(\bar{v}_{i})\right)$. A formula $\text{Inv}$ is a $k$-inductive invariant iff $\text{Inv}$ is a $(k - 1)$-invariant and is inductive after $k$ steps of $T$, i.e., the following formula is valid: $\text{Tr}[\text{Inv}]^{k} \Rightarrow \text{Inv}(\bar{v}_{k})$. Compared to simple induction, $k$-induction strengthens the hypothesis in the induction step: $\text{Inv}$ is assumed to hold between steps 0 to $k - 1$ and is established in step $k$. Whenever $\text{Inv} \Rightarrow P$, we say that $\text{Inv}$ 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 $\text{Inv}_k$ there exists a 1-inductive strengthening $\text{Inv}_1$ such that $\text{Inv}_1 \Rightarrow \text{Inv}_k$. Theoretically $\text{Inv}_1$ might be exponentially bigger than $\text{Inv}_k$ [6]. In practice, both invariants tend to be of similar size.

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

Craig Interpolation [10]. We use an extension of Craig Interpolants to sequences, which is common in Model Checking. Let $A = [A_1, \ldots, A_N]$ such that $A_1 \land \cdots \land A_N$ is unsatisfiable. A sequence interpolant $I = \text{SEQITP}(A)$ for $A$ is a sequence of formulas $I = [I_2, \ldots, I_N]$ such that (a) $A_1 \Rightarrow I_2$, (b) $\forall 1 < i < N \cdot I_i \land A_i \Rightarrow I_{i+1}$, (c) $I_N \land A_N \Rightarrow \bot$, and (d) $I_i$ is over variables that are shared between the corresponding prefix of $A$.

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 $T = (\bar{v}, \text{Init}, \text{Tr}, \text{Bad})$.

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 $F = [F_0, \ldots, F_N]$ that satisfy the following two properties:

$$\text{Init}(\bar{v}) = F_0(\bar{v}) \quad \forall 0 \leq i < N \cdot F_i(\bar{v}) \land \text{Tr}(\bar{v}, \bar{v}') \Rightarrow F_{i+1}(\bar{v}')$$

(4)

An element $F_i$ of a trace is called a frame. The index of a frame is called a level. $F$ 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 $F = [F_0, \ldots, F_N]$ is $|F| = N$. For $k \leq N$, we write $F^k = [F_k, \ldots, F_N]$ for the $k$-suffix of $F$.

A trace $F$ of size $N$ is stronger than a trace $G$ 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: $\forall i \cdot F_i \Rightarrow \neg \text{Bad}$; 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 $F = [F_0, \ldots, F_N]$ as :

$$\text{Tr}[F^k] = \bigwedge_{i=k}^{N} F_i(\bar{v}_i) \land \text{Tr}(\bar{v}_i, \bar{v}_{i+1})$$

(5)
We write $\text{Tr}[F]$ to denote an unrolling of a 0-suffix of $F$ (i.e. $F$ itself). Intuitively, $\text{Tr}[F^k]$ is satisfiable iff there is a $k$-step execution of the $\text{Tr}$ that is consistent with the $k$-suffix $F^k$. If a transition system $T$ admits a safe trace $F$ of size $|F| = N$, then $T$ does not admit counterexamples of length less than $N$. A safe trace $F$, with $|F| = N$ is extendable with respect to level $0 \leq i \leq N$ iff there exists a safe trace $G$ stronger than $F$ such that $|G| > N$ and $F_i \land \text{Tr} \Rightarrow G_{i+1}$. $G$ and the corresponding level $i$ are called an extension trace and an extension level of $F$, respectively. SAT-based model checking algorithms work by iteratively extending a given safe trace $F$ 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 $F$ by $W(F)$. The existence of an extension level $i$ implies that an unrolling of the $i$-suffix does not contain any Bad states:

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

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

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 [7,13,29].

Pdr maintains a monotone, clausal trace $F$ with $\text{Init}$ as the first frame ($F_0$). The trace $F$ is extended by recursively computing and blocking (if possible) states that can reach $\text{Bad}$ (called bad states). A bad state is blocked at the largest level possible. Algorithm 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 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 $d$. 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 $G$ is an extension trace of $F$ 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.
when the trace is closed.

A helper routine for Example 2. First, we introduce a concept of extending a trace using relative induction. Second, we present

Algorithm 1. PdrBlock.

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

1. \(w \leftarrow N + 1; G \leftarrow F; Q.\text{push}((\text{Bad}, N + 1))\)
2. \(\text{while } \neg Q.\text{empty}() \text{ do}\)
3. \(\langle s, d \rangle \leftarrow Q.\text{pop}()\)
4. \(\text{if } d = 0 \text{ then return } []\)
5. \(\text{if } \neg \text{isSat}(F_{d-1}(\bar{v}) \land Tr(\bar{v}) \land s(\bar{v})) \text{ then}\)
6. \(t \leftarrow \text{predecessor}(s)\)
7. \(Q.\text{push}(t, d - 1)\)
8. \(Q.\text{push}(s, d)\)
9. \(\text{else}\)
10. \(\forall 0 \leq i \leq d \cdot G_i \leftarrow (G_i \land \text{PdrIndGen}(\neg s))\)
11. \(w \leftarrow \min(w, d)\)
12. \(\text{return } G\)

Algorithm 2. Avy.

\[\text{Input: A transition system } T = (\text{Init}, Tr, Bad)\]
\[\text{Output: SAFE/UNSAFE}\]
1. \(F_0 \leftarrow \text{Init}; N \leftarrow 0\)
2. \(\text{repeat}\)
3. \(\text{if } \neg \text{isSat}(Tr[F] \land Bad(\bar{v}_{N+1})) \text{ then}\)
4. \(\text{return } \text{UNSAFE}\)
5. \(k \leftarrow \max\{i | \neg \text{isSat}(Tr[F] \land Bad(\bar{v}_{N+1}))\}\)
6. \(I_k, \ldots, I_{N+1} \leftarrow \text{seqTP}(Tr[F] \land Bad(\bar{v}_{N+1}))\)
7. \(\forall 0 \leq i \leq k \cdot G_i \leftarrow F_i\)
8. \(\forall k \leq i \leq (N+1) \cdot G_i \leftarrow F_i \land I_i\)
9. \(F \leftarrow \text{AvyMkTrace}([G_0, \ldots, G_{N+1}])\)
10. \(F \leftarrow \text{PdrPush}(F)\)
11. \(\text{if } \exists 1 \leq i \leq N \cdot F_i \Rightarrow \left(\bigvee_{j=0}^{i-1} F_j\right) \text{ then}\)
12. \(\text{return } \text{SAFE}\)
13. \(N \leftarrow N + 1\)
14. \(\text{until } \infty\)

Avy, shown in Algorithm 2, is an alternative to Pdr that combines interpolation and recursive blocking. Avy starts with a trace \(F\), with \(F_0 = \text{Init}\), that is extended in every iteration of the main loop. A counterexample is returned whenever \(F\) is not extendable (line 3). Otherwise, a sequence interpolant is extracted from the unsatisfiability of \(Tr[F_{max}(\bar{v})] \land Bad(\bar{v}_{N+1})\). A longer trace \(G = [G_0, \ldots, G_N, G_{N+1}]\) is constructed using the sequence interpolant (line 7). Observe that \(G\) is an extension trace of \(F\). While \(G\) is safe, it is neither monotone nor clausal. A helper routine AvyMkTrace is used to convert \(G\) 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 \(F\), with \(|F| = N\), 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 \(G\) stronger than \(F\) such that \(|G| > N + 1\) 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 \(G\) as an \((i, k)\)-extension trace, or simply a strong extension trace (SET) when \((i, k)\) is not important. Note that for \(k = 1\), \(G\) is just an extension trace.

Example 2. For Fig. 1, the trace \(F = [c = 0, c < 66]\) is strongly extendable at level 1. A valid \((1, 2)\)-extension trace is \(G = [c = 0, (c \neq 65) \land (c < 66), c < 66]\). Note that \((c < 66)\) is 2-inductive relative to \(F_1\), i.e., \(Tr[F_1]^2 \Rightarrow (c'' < 66)\).
We write $K(F)$ for the set of all SELs of $F$. We define an order on SELs by:

$$(i_1, k_1) \preceq (i_2, k_2) \text{ iff (i) } i_1 < i_2; \text{ or (ii) } i_1 = i_2 \land k_1 > k_2.$$ 

The maximal SEL is $\max(K(F))$.

### Algorithm 3. kAVY algorithm.

**Input:** A transition system $T = (\text{Init}, \text{Tr}, \text{Bad})$

**Output:** SAFE/UNSAFE

1. $F \leftarrow [\text{Init}]; N \leftarrow 0$
2. repeat
   
   // Invariant: $F$ is a monotone, clausal, safe, inductive trace
   
   3. $U \leftarrow \text{Tr}[F^0] \land \text{Bad}(\bar{v}_{N+1})$
   
   4. if $\text{isSat}(U)$ then return UNSAFE
   
   5. $(i, k) \leftarrow \max\{(i, k) \mid \neg\text{isSat}(\text{Tr}[F^i]^k \land \text{Bad}(\bar{v}_{N+1}))\}$
   
   6. $[F_0, \ldots, F_{N+1}] \leftarrow \text{kAVYExtend}(F, (i, k))$
   
   7. $[F_0, \ldots, F_{N+1}] \leftarrow \text{PdrPush}([F_0, \ldots, F_{N+1}])$
   
   8. if $\exists 1 \leq i \leq N : F_i \Rightarrow \left( \bigvee_{j=0}^{i-1} F_j \right)$ then return SAFE
   
   9. $N \leftarrow N + 1$
3. until $\infty$

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 $\text{Tr}[F^i]^k$ to denote this characteristic formula for SEL $(i, k)$:

$$\text{Tr}[F^i]^k = \begin{cases} 
\text{Tr}[F^i_{i+1,k-1}] \land \text{Tr}[F^i+1] & \text{if } 0 \leq i < N \\
\text{Tr}[F^N_{N+1,k-1}] & \text{if } i = N 
\end{cases} \tag{6}$$

**Proposition 2.** Let $F$ be a safe trace, where $|F| = N$. Then, $(i, k), 1 \leq k \leq i+1 \leq N+1$, is an SEL of $F$ iff the formula $\text{Tr}[F^i]^k \land \text{Bad}(\bar{v}_{N+1})$ is unsatisfiable.

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

**Lemma 1.** Let $(i, k) = \max(K(F))$, then $i \geq \max(W(F))$.

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 $F = [\text{Init}]$ and iteratively extends $F$ 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 Sect. 4.2). Then, it constructs a strong extension trace using kAVYExtend (line 6) (described in Sect. 4.1). Finally, PdrPush is called to check whether the trace is closed. Note that $F$ 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 \textsc{kAVYEXTEND} (shown in Algorithm 4) that given a trace \( F \) of size \(|F| = N\) and an \((i, k)\) SEL of \( F \) constructs an \((i, k)\)-extension trace \( G \) of size \(|G| = N + 1\). 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 \( F \) and its SEL \((i, k)\).

**Sequence Interpolation for SEL.** Let \((i, k)\) be an SEL of \( F \). By Proposition 2, \( \Psi = Tr[\Phi]^k \land \text{Bad}(\bar{v}_{N+1}) \) is unsatisfiable. Let \( \mathcal{A} = \{A_{i-k+1}, \ldots, A_{N+1}\} \) be a partitioning of \( \Psi \) defined as follows:

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

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

\[
\begin{align*}
F_i \land Tr &\Rightarrow I'_{i-k+2} \quad \forall i - k + 2 \leq j \leq i \cdot (F_i \land I_j) \land Tr \Rightarrow I'_{j+1} \quad (\heartsuit) \\
I_{N+1} &\Rightarrow \neg \text{Bad} \quad \forall i < j \leq N \cdot (F_j \land I_j) \land Tr \Rightarrow I'_{j+1}
\end{align*}
\]

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 \( I \) is \( k \)-inductive relative to the frame \( F_i \):

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

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

We use Lemma 2 to define a strong extension trace \( G \):

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

\[
G_j = \begin{cases} 
F_j & \text{if } 0 \leq j < i - k + 2 \\
F_j \land \left( \bigwedge_{m=i-k+2}^{j} I_m \right) & \text{if } i - k + 2 \leq j < i + 2 \\
(F_j \land I_j) & \text{if } i + 2 \leq j < N + 1 \\
I_{N+1} & \text{if } j = N + 1
\end{cases}
\]
Then, $G$ is an $(i,k)$-extension trace of $F$ (not necessarily monotone).

Proof. By Lemma 2, $G_{i+1}$ is $k$-inductive relative to $F_i$. Therefore, it is sufficient to show that $G$ is a safe inductive trace that is stronger than $F$. By definition, $\forall 0 \leq j \leq N \cdot G_j \Rightarrow F_j$. By ($\forall$), $F_i \land Tr \Rightarrow I'_{j+k+2}$ and $\forall i - k + 2 \leq j < i + 2 \cdot (F_i \land I_j) \land Tr \Rightarrow I'_{j+1}$. By induction over $j$, $((F_i \land \bigwedge_{m=i-k+2}^j I_m) \land Tr) \Rightarrow \bigwedge_{m=i-k+2}^{j+1} I'_m$ for all $i - k + 2 \leq j < i + 2$. Since $F$ is monotone, $\forall i - k + 2 \leq j < i + 2$. By ($\forall$), $\forall i < j \leq N \cdot (F_j \land I_j) \land Tr \Rightarrow I'_{j+1}$. Again, since $F$ is a trace, we conclude that $\forall i < j < N \cdot (F_j \land I_j) \land Tr \Rightarrow (F_{j+1} \land I_{j+1})'$. Combining the above, $G_j \land Tr \Rightarrow G'_{j+1}$ for $0 \leq j \leq N$. Since $F$ is safe and $I_{N+1} \Rightarrow \neg Bad$, then $G$ is safe and stronger than $F$. \hfill $\Box$

Lemma 3 defines an obvious procedure to construct an $(i,k)$-extension trace $G$ for $F$. However, such $G$ 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 $F$.

The procedure KAVYEXTEND is shown in Algorithm 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 $A$. The extension trace $G$ is initialized to $F$ 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 $I$; Phase 2 (line 8) computes $G_{i+1}$ using $I_{i+1}$; Phase 3 (lines 9–12) computes the suffix $G_{i+2}$ using the last $(N - i)$ elements of $I$. 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 Algorithm 4. We present each phase in turn.

Recall that PDRBLOCK takes a trace $F$ (that is safe up to the last frame) and a transition system, and returns a safe strengthening of $F$, while ensuring that the result is monotone and clausal. This guarantee is maintained by Algorithm 4, by requiring that any clause added to any frame $G_i$ of $G$ 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 $I$. Note that in that phase, new clauses are always added to frame $G_{i+1}$, and all frames before it!
Algorithm 4. \textsc{kAVyExtend}. The invariants marked $\dagger$ hold only when the PDR\textsc{Block} does no inductive generalization.

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{Input:} a monotone, clausal, safe trace $F$ of size $N$
\State \textbf{Input:} A strong extension level $(i, k)$ s.t. $\text{Tr}[F_i]^k \land \mathit{Bad}(\overline{v}_{N+1})$ is unsatisfiable
\State \textbf{Output:} a monotone, clausal, safe trace $G$ of size $N + 1$
\State $I_{i-k+2}, \ldots, I_{N+1} \leftarrow \text{SEQITP}(\text{Tr}[F_i]^k \land \mathit{Bad}(\overline{v}_{N+1}))$
\State $G \leftarrow [F_0, \ldots, F_N, \top]$
\For{$j \leftarrow i - k + 1$ \KwTo $i$}
\State $P_j \leftarrow (G_j \lor (G_{i+1} \land I_{j+1}))$
\hspace{1em} // \textbf{Inv}_1: $G$ is monotone and clausal
\hspace{1em} // \textbf{Inv}_2: $G_i \land \text{Tr} \Rightarrow P_j$
\hspace{1em} // \textbf{Inv}_3: $\forall j < m \leq (i + 1) \cdot G_m \equiv F_m \land \bigwedge_{i=k+1}^{i-1} (G_{i} \lor I_{i+1})$
\hspace{1em} // \textbf{Inv}_4: $\forall j < m \leq (i + 1) \cdot G_m \Rightarrow F_m \land \bigwedge_{i=k+1}^{i-1} (G_{i} \lor I_{i+1})$
\State $[\ldots, G_{i+1}] \leftarrow \text{PDR\textsc{Block}}([\text{Init}, G_i, G_{i+1}], (\text{Init}, \text{Tr}, \neg P_j))$
\EndFor
\If{$i = 0$}
\State $[\ldots, G_{i+1}] \leftarrow \text{PDR\textsc{Block}}([\text{Init}, G_{i+1}], (\text{Init}, \text{Tr}, \neg P_i))$
\Else
\State $[\ldots, G_{i+1}] \leftarrow \text{PDR\textsc{Block}}([\text{Init}, G_i, G_{i+1}], (\text{Init}, \text{Tr}, \neg P_i))$
\EndIf
\For{$j \leftarrow i + 1$ \KwTo $N + 1$}
\State $P_j \leftarrow G_j \lor (G_{j+1} \land I_{j+1})$
\hspace{1em} // \textbf{Inv}_5: $G_j \land \text{Tr} \Rightarrow P_j$
\State $[\ldots, G_{j+1}] \leftarrow \text{PDR\textsc{Block}}([\text{Init}, G_j, G_{j+1}], (\text{Init}, \text{Tr}, \neg P_j))$
\EndFor
\State $G \leftarrow \text{PDR\textsc{Push}}(G)$
\hspace{1em} // \textbf{Inv}_6: $G$ is an $(i, k)$-extension trace of $F$
\hspace{1em} // \textbf{Inv}_7: $G$ is an extension trace of $F$
\State \Return $G$
\end{algorithmic}
\end{algorithm}

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

$\textbf{Inv}_2$ ensures that the trace given to PDR\textsc{Block} at line 5 can be made safe relative to $P_j$. From the post-condition of PDR\textsc{Block}, it follows that at iteration $j$, $G_{i+1}$ is strengthened to $G_{i+1}^*$ such that $G_{i+1}^* \Rightarrow P_j$ and $G$ 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 PDR\textsc{Block} in this phase do not satisfy an expected pre-condition: the frame $G_i$ in $[\text{Init}, G_i, G_{i+1}]$ might not be safe for property $P_j$. However, we can see that $\text{Init} \Rightarrow P_j$ and from $\textbf{Inv}_2$, it is clear that $P_j$ is inductive relative to $G_i$. This is a sufficient precondition for PDR\textsc{Block}. 


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 $\text{Inv}_6$ 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 $G$ is a monotone clausal trace. The invariant implies that at the end of the loop $G_{N+1} \Rightarrow G_N \lor I_{N+1}$, making $G$ safe. Thus, at the end of the loop $G$ is a safe monotone clausal trace that is stronger than $F$. What remains is to show is that $G_{i+1}$ is $k$-inductive relative to $F_i$.

Let $\varphi$ be the formula from Lemma 2. Assuming that PDrBlock did no inductive generalization, Phase 1 maintains $\text{Inv}_3^\dagger$, which states that at iteration $j$, PDrBlock strengthens frames $\{G_m\}$, $j < m \leq (i + 1)$. $\text{Inv}_3^\dagger$ holds on loop entry, since initially $G = F$. 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, $G^*_m \equiv G_m \land P_j$. Since $G_j \Rightarrow G_m$, this simplifies to $G^*_m \equiv G_m \land (G_j \lor I_{j+1})$. Expanding $G_m$, we get $G^*_m \equiv F_m \land \bigwedge_{\ell=i-k+1}^j (G_{\ell} \lor I_{\ell+1})$. Thus, $\text{Inv}_3^\dagger$ holds at the end of the loop.

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

Theorem 2. Given a safe trace $F$ of size $N$ and an SEL $(i, k)$ for $F$, kAVYExtend returns a clausal monotone extension trace $G$ of size $N+1$. Furthermore, if PDrBlock does no inductive generalization then $G$ 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 $G$ that is a monotone clausal extension of $F$. However, $G$ might be stronger than any $(i, k)$-extension of $F$. The invariants marked with $\dagger$ 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 Algorithm 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

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

Note that \(k\) depends on \(i\). For a SEL \((i, k) \in \mathcal{K}(F)\), we refer to the formula \(\text{Tr}[F]^{i}\) 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 Algorithm 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. Algorithm 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 \((j, \ell) \not\in \mathcal{K}(F)\), then \(\forall p > j \cdot (p, \ell) \not\in \mathcal{K}(F)\). This is used at line 7 to increase the depth of induction once it is known that \((j, \ell) \not\in \mathcal{K}(F)\). On the other hand, if \((j, \ell) \in \mathcal{K}(F)\), there might be a larger SEL \((j + 1, \ell)\). Thus, whenever a SEL \((j, \ell)\) 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 \(\ell\) is incremented only when there does not exist a larger SEL with the current value of \(\ell\). 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 \((i, k) = \max \mathcal{K}(F)\).
In the worst case, Algorithm 6 makes at most $3N$ \textsc{sat} queries. However, compared to Algorithm 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.

5 Evaluation

We implemented \textsc{kAvy} on top of the \textsc{Avy Model Checker}\footnote{All code, benchmarks, and results are available at \url{https://arieg.bitbucket.io/avy/}.}. For line 5 of Algorithm 3 we used Algorithm 5. We evaluated \textsc{kAvy}'s performance against a version of \textsc{Avy} [29] from the Hardware Model Checking Competition 2017 [5], and the \textsc{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 8 GB RAM limit and 30 min 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., \textit{intel}) include benchmarks that have not been included in any competition. The first column in Table 1 indicates the category. \textbf{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, \textsc{kAvy} solves 14 instances in \textit{oc8051} that are not solved by any other solver. The \textsc{VBS} column indicates the \textit{Virtual Best Solver}—the result of running all the three solvers in parallel and stopping as soon as one solver terminates successfully.

Overall, \textsc{kAvy} solves more \textit{safe} instances than both \textsc{Avy} and \textsc{Pdr}, while taking less time than \textsc{Avy} (we report time for solved instances, ignoring time-outs). The \textsc{VBS} column shows that \textsc{kAvy} is a promising new strategy, significantly improving overall performance. In the rest of this section, we analyze the
Table 1. Summary of instances solved by each tool. Timeouts were ignored when computing the time column.

<table>
<thead>
<tr>
<th>BENCHMARKS</th>
<th>KAVY</th>
<th>AVY</th>
<th>PDR</th>
<th>VBS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAFE</td>
<td>UNSAFE</td>
<td>Time(m)</td>
<td>SAFE</td>
</tr>
<tr>
<td>HWMCC’ 17</td>
<td>137 (16)</td>
<td>38</td>
<td>499</td>
<td>128 (3)</td>
</tr>
<tr>
<td>HWMCC’ 15</td>
<td>193 (4)</td>
<td>84</td>
<td>412</td>
<td>191 (3)</td>
</tr>
<tr>
<td>HWMCC’ 14</td>
<td>49</td>
<td>27 (1)</td>
<td>124</td>
<td>58 (4)</td>
</tr>
<tr>
<td>intel</td>
<td>32 (1)</td>
<td>9</td>
<td>196</td>
<td>32 (1)</td>
</tr>
<tr>
<td>6s</td>
<td>73 (2)</td>
<td>20</td>
<td>157</td>
<td>81 (4)</td>
</tr>
<tr>
<td>nusmv</td>
<td>13</td>
<td>0</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>bob</td>
<td>30</td>
<td>5</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td>pdt</td>
<td>45</td>
<td>1</td>
<td>54</td>
<td>45 (1)</td>
</tr>
<tr>
<td>oski</td>
<td>26</td>
<td>89 (1)</td>
<td>174</td>
<td>28 (2)</td>
</tr>
<tr>
<td>beem</td>
<td>10</td>
<td>1</td>
<td>49</td>
<td>10</td>
</tr>
<tr>
<td>oc8051</td>
<td>34 (14)</td>
<td>0</td>
<td>286</td>
<td>20</td>
</tr>
<tr>
<td>power</td>
<td>4</td>
<td>0</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>shift</td>
<td>5 (2)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>necla</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>7 (1)</td>
</tr>
<tr>
<td>prodcell</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bc57</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>326 (19)</td>
<td>141 (1)</td>
<td>957</td>
<td>319 (8)</td>
</tr>
</tbody>
</table>

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 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 e. 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 f, 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-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$^2$, 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 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 Sect. 4.2. Deciding when and how much to settle for remains a challenge.

---

2 We used the $k$-induction engine ind in ABC [8].
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 Engineering Research Council Canada.

References


Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.