Interpolating Property Directed Reachability[⋆]

Yakir Vizel[1] and Arie Gurfinkel[2]

[1] Computer Science Department, The Technion, Haifa, Israel
[2] Carnegie Mellon Software Engineering Institute, Pittsburgh, USA

Abstract. Current SAT-based Model Checking is based on two major approaches: Interpolation-based (Imc) (global, with unrollings) and Property Directed Reachability/IC3 (Pdr) (local, without unrollings). Imc generates candidate invariants using interpolation over an unrolling of a system, without putting any restrictions on the SAT-solver's search. Pdr generates candidate invariants by a local search over a single instantiation of the transition relation, effectively guiding the SAT solver's search. The two techniques are considered to be orthogonal and have different strength and limitations. In this paper, we present a new technique, called Avy, that effectively combines the key insights of the two approaches. Like Imc, it uses unrollings and interpolants to construct an initial candidate invariant, and, like Pdr, it uses local inductive generalization to keep the invariants in compact clausal form. On the one hand, Avy is an incremental Imc extended with a local search for CNF interpolants. On the other, it is Pdr extended with a global search for bounded counterexamples. We implemented the technique using ABC and have evaluated it on the HWMCC benchmark-suite from 2012 and 2013. Our results show that the prototype significantly outperforms Pdr and McMillan's interpolation algorithm (as implemented in ABC) on the industrial sub-category of the benchmark.

1 Introduction

SAT-based (unbounded) Model Checking (MC) is an extremely successful technique for both Hardware [12,4,10] and Software [13,2,11] verification. Current state-of-the-art techniques are Interpolation-based Model Checking (Imc) [12,15] and Property Directed Reachability/IC3 (Pdr) [4,10]. Pdr and Imc are able to either verify a property by generating a safe inductive invariant, or falsify a property by finding a counterexample. Conceptually, both work by repeatedly *generalizing* bounded proofs of correctness, until either a safe inductive invariant is synthesized or a counterexample is found. They scale to systems with an enormous number of states, are considered orthogonal, and have different strength and weaknesses.

Imc works by searching for a counterexample via repeatedly posing Bounded Model Checking [3] (BMC) queries to a SAT-solver. If a BMC query Q is satisfied, a counterexample is found. Otherwise, the SAT-solver generates a proof of

[⋆] This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense. This material has been approved for public release and unlimited distribution. DM-0001263.

unsatisfiability of Q. An interpolation procedure is then used to generalize the proof to a candidate safe invariant using sequence interpolants [15]. If the invariant is also inductive (checked by an additional SAT query), the procedure stops and returns SAFE to the user, indicating the validity of the checked property. Otherwise, the process repeats with another, longer, BMC query.

IMC leverages both advances in BMC and in interpolation. It can be seen as a simple addition to BMC that turns it into a complete Model Checking procedure. Other than proof-logging which is necessary for interpolation, it poses no restrictions on the SAT-solver's search. However, IMC does not offer much control over generalization. It is at the mercy of both the SAT-solver that provides a particular resolution proof, and of the procedure used to generate the interpolant. For example, attempts to improve IMC by using interpolation algorithms with different strength have not been very successful [9]. Furthermore, the interpolants tend to be large, which poses additional limitation on their use.

PDR is similar to IMC, but approaches the process in a completely different manner. Instead of blindly relying on the SAT-solver, it manages both the search for the counterexample and the generalization phases. Conceptually, PDR is based on a backward search. Starting with a bad (UNSAFE) state, it uses a SAT-solver to repeatedly find a one-step predecessor state. Thus, all SAT-queries are local, involving only one instance of the transition relation, and no BMC-unrolling is used. If the bad suffix can be extended all the way to the initial state, a counterexample is found. Otherwise, when a suffix cannot be extended further, a process called *inductive generalization* [4], is used to learn a consequence that blocks the current suffix (and possibly many others). The conjunction of all such learned consequences is used to synthesize an inductive invariant. While this description omits many important aspects of PDR, it is sufficient for now.

PDR offers many advantages compared to IMC, including incremental solving and fine-grained control over generalization. However, it is limited to a fixed *local* search strategy that can be inefficient. In fact, it is not difficult to construct examples in which backward search is ineffective and PDR does not perform well.

In this paper, we present a new algorithm, AVY, that strives to overcome these deficiencies by combining both *global* interpolant-driven generalization with *local* inductive generalization. AVY can be seen as a combination of PDR and IMC. On the one hand, it extends IMC with PDR-like local reasoning in the form of local search and inductive generalization. On the other hand, it extends PDR with the use of unrolling and proof-based interpolation. More interestingly, it allows the combination of IMC and PDR strategies inside a single solver.

The first step of AVY is similar to IMC: it unrolls the system and searches for a counterexample. If none is found, it generates a candidate invariant using sequence interpolants [15]. This is the global generalization phase. Next, it enters the local generalization phase and uses PDR-style inductive generalization to strengthen the candidate invariant and to put it into CNF. If the candidate is inductive, the process stops. Otherwise, the next global phase is entered.

Maintaining the candidate invariant in CNF allows AVY to use it as "learned clauses" in the next global phase. When a new global phase starts, AVY adds the clauses from the previously computed candidate invariant into the checked BMC formula, thus making the global phase incremental. This significantly reduces the search space for the SAT-solver to explore. It also reduces the size of the resulting resolution proof and the computed interpolants. This addresses the main problem with IMC: lack of incrementality as already learned interpolants are not used in successive iterations and interpolant growth.

Adding the learned clauses to the BMC problem at a given iteration N, makes it, in a way, equivalent to the problem PDR tries to solve at iteration N. Though, unlike PDR, AVY handles this problem globally, with one SAT-solver instance that can roam over the entire search space, and does not break it to local checks as part of a backward search. This kind of strategy addresses the main weakness of PDR: no use of "global" knowledge during the search.

The combination of interpolation and inductive reasoning allows AVY to benefit from the advantages of both methods. It uses the SAT-solver without guiding it during the search, but it does guide its proof construction. The advantage of this combination is evident in our experiments. We have implemented AVY on top of ABC [5] and compared it against PDR and McMillan's interpolation (ITP), as implemented in ABC, on the HWMCC'12/13 benchmarks. Our experiments indicate that AVY can solve a considerable number of test cases, especially on the industrial sub-category, that are not solved by either PDR nor ITP.
Related work. This paper builds on Interpolation-based Model Checking [12,15], IC3 [4], and PDR [10]. We describe them in detail in Section 3. Some of the techniques used in AVY have appeared before, but not in the way AVY combines them. Like [6], we use sequence interpolants, but we show that they can be more efficient than the original algorithm in [12]. Like [1], we re-use previously computed interpolants, but we combine re-use with inductive generalization. Our approach can be seen as an efficient extension of [7] to sequence interpolation.

As stated above, AVY is a synergy between an interpolation-based approach and PDR. Ideas for combining the two have also appeared in [16,17]. In [16], the authors suggest to use both forward and backward reachable sets of states. This allows them to try and block a set of all bad states in a local manner that resembles the blocking of a bad state applied by PDR. Unlike [16], in this work we only use the forward reachable states that are derived by means of interpolation, and use specific PDR functionality to transform these sets into CNF and use them to simplify the successive BMC invocations. In [17], the authors show how to compute interpolants in CNF and create a variant of the algorithm that appears in [12], which uses the fact that interpolants are in CNF in order to apply PDR-style reasoning. There are two major differences between AVY and the approach that appear in [17]. First, in [17], the resolution refutation is used to derive a "near interpolant" in CNF, which is then strengthened and transformed into an interpolant by applying inductive generalization on the (A, B) pair, while AVY derives a sequence interpolant, and then uses PDR to transform it to CNF. Second, like in [17], AVY also uses the fact that interpolants are in CNF and tries to push clauses between different interpolants. But, while [17] uses pushing only to learn clauses that may appear in later interpolants that were not computed yet, AVY, as stated, uses the pushed clauses to simplify the BMC formula.

The rest of the paper is structured as follows. After describing the necessary background and notation in Section 2, we give an overview of SAT-based Model Checking in Section 3. Section 4 presents two versions of Avy, a basic and an optimized one. We describe our experimental results in Section 5, and conclude in Section 6.

2 Preliminaries

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

Safety verification. A transition system T is a tuple $(\mathcal{V}, \mathit{Init}, \mathit{Tr}, \mathit{Bad})$, where $\mathcal{V}$ is a set of variables that defines the states of the system (i.e., $2^{\mathcal{V}}$), Init and Bad are formulas with variables in $\mathcal{V}$ denoting the set of initial states and bad states, respectively, and Tr is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$, denoting the transition relation. A state $s \in 2^{\mathcal{V}}$ is said to be reachable in T if and only if (iff) there exists a state $s_0 \in \mathit{Init}$, and $(s_i, s_{i+1}) \in \mathit{Tr}$ for $0 \leq i \leq N$, and $s = s_N$.

A transition system T is UNSAFE iff there exists a state $s \in \mathit{Bad}$ s.t. s is reachable. Equivalently, T is UNSAFE iff there exists a number N such that the following formula is satisfiable:

$$\mathit{Init}(v_0) \wedge \left(\bigwedge_{i=0}^{N-1} \mathit{Tr}(v_i, v_{i+1}) \right) \wedge \mathit{Bad}(v_N) \tag{1}$$

When T is UNSAFE and $s_N \in \mathit{Bad}$ is the reachable state, the path from $s_0 \in \mathit{Init}$ to s_N is called a *counterexample* (CEX).

A transition system T is SAFE iff all reachable states in T do not satisfy Bad. Equivalently, there exists a formula Inv, called an *inductive safe invariant*[3], that satisfies:

$$\mathit{Init}(v) \to \mathit{Inv}(v) \qquad \mathit{Inv}(v) \wedge \mathit{Tr}(v, u) \to \mathit{Inv}(u) \qquad \mathit{Inv}(v) \to \neg \mathit{Bad}(v) \tag{2}$$

A *safety* verification problem is to decide whether a transition system T is SAFE or UNSAFE, i.e., whether there exists an initial state in Init that can reach a bad state in Bad, or synthesize a safe inductive invariant.

In SAT-based model checking, the verification problem is determined by computing over-approximations of the states reachable in T and, by that, trying to either construct an invariant or find a CEX.

Craig Interpolation. Given a pair of inconsistent formulas (A, B) (i.e., $A \wedge B \models \bot$), a *Craig interpolant* [8] for (A, B) is a formula I such that:

$$A \to I \qquad\qquad I \to \neg B \qquad\qquad \mathcal{L}(I) \subseteq \mathcal{L}(A) \cap \mathcal{L}(B) \tag{3}$$

[3] The reachable states form an inductive invariant. The inductive invariant is *safe* if the reachable states do not intersect the bad states.

where $\mathcal{L}(A)$ denotes the set of all atomic propositions in A. A *sequence (or path) interpolant* extends interpolation to a sequence of formulas. We write $\boldsymbol{F} = [F_1, \ldots, F_N]$ to denote a sequence with N elements, and $\boldsymbol{F}_i$ for the ith element of the sequence. Given an unsatisfiable sequence of formulas $\boldsymbol{A} = [A_1, \ldots, A_N]$, i.e., $A_1 \wedge \cdots \wedge A_N \models \bot$, a *sequence interpolant* $\boldsymbol{I} = \textsc{seqItp}(\boldsymbol{A})$ for $\boldsymbol{A}$ is a sequence of formulas $\boldsymbol{I} = [I_1, \ldots, I_{N-1}]$ such that:

$$A_1 \to I_1 \qquad \forall 1 < i < N \cdot I_{i-1} \wedge A_i \to I_i \qquad I_{N-1} \wedge A_N \to \bot \tag{4}$$

and for all $1 \leq i \leq N$, $\mathcal{L}(I_i) \subseteq \mathcal{L}(A_1 \wedge \cdots \wedge A_i) \cap \mathcal{L}(A_{i+1} \wedge \cdots \wedge A_N)$. We use subscripts on brackets to mark interpolation partitions for a formula. For example, $(A)_0 \wedge (B)_1 \wedge (C)_0$ means that A and C belong to partition 0 and B to partition 1, respectively.

3 SAT-Based Model Checking

In this section, we review two algorithms for SAT-based *unbounded* Model Checking – Interpolation-based Model Checking (IMC), and Property Directed Reachability/IC3 (PDR).

The key insight in both algorithms is to maintain an over-approximation of a set of reachable states in an inductive trace. An *inductive trace*, or simply a trace, is a sequence of formulas $[F_0, \ldots, F_N]$ that satisfy:

$$\mathit{Init} \to F_0 \qquad \forall 0 \leq i < N \cdot F_i(v) \wedge \mathit{Tr}(v, u) \to F_{i+1}(u) \tag{5}$$

A trace is *safe* if each F_i is safe: $\forall i \cdot F_i \to \neg \mathit{Bad}$; it is *monotone* if $\forall 0 \leq i < N \cdot F_i \to F_{i+1}$; it is *clausal* if each F_i is in CNF (in this case, we often abuse notation and treat each F_i as a set of clauses). A trace $[F_0, \ldots, F_N]$ is *stronger* than a trace $[G_0, \ldots, G_N]$ if $\forall 0 \leq i \leq N \cdot F_i \to G_i$. We assume that traces are silently extended as needed, by letting $F_i = \top$ for all $i > N$ for any trace $[F_0, \ldots, F_N]$. Traces are closed under pointwise conjunction.

A trace $[F_0, \ldots, F_N]$ is *closed* if $\exists 1 \leq i \leq N \cdot F_i \to \left(\bigvee_{j=0}^{i-1} F_j\right)$. There is an obvious relationship between existence of closed traces and safety of a transition system:

Theorem 1. *A transition system T is SAFE iff it admits a safe closed trace.*

Thus, safety verification is reduced to searching for a safe closed trace or finding a CEX.

3.1 Interpolation-Based Model Checking

The original interpolation-based algorithm is due to McMillan [12]. Here, we present its variant from [15], called Imc, based on sequence interpolants. This version is closer to Pdr (described in Section 3.2) and is a basis for our algorithm.

Imc is shown in Alg. 1. It maintains a trace $[F_0, \ldots, F_N]$. The trace is made safe toward the end of the loop (line 5). In the beginning of each iteration,

```
  Input: Transition system $T = (Init, Tr, Bad)$
1 $F_0 \leftarrow Init\ ;\ N \leftarrow 0$
2 repeat
3     $\boldsymbol{G} \leftarrow \textsc{ImcMkSafe}([F_0, \ldots, F_N], Bad)$
4     if $\boldsymbol{G} = [\,]$ then return UNSAFE;
5     $\forall 0 \leq i \leq N \cdot F_i \leftarrow \boldsymbol{G}[i]$
      // Invariant: $F_0, \ldots, F_N$ is a safe trace
6     if $\exists 1 \leq i \leq N \cdot F_i \to (\bigvee_{j=0}^{i-1} F_j)$ then return SAFE;
7     $N \leftarrow N + 1\ ;\ F_N \leftarrow \top$
8 until $\infty$;
```

Algorithm 1: IMC.

```
  Input: Transition system $T = (Init, Tr, Bad)$
  Input: A trace $F_0, \ldots, F_N$
1 $\varphi \leftarrow (Init(v_0))_0 \wedge \bigwedge_{i=0}^{N-1} (Tr(v_i, v_{i+1}))_i \wedge (Bad(v_N))_N$
2 if $\textsc{isSat}(\varphi)$ then return $[\,]$;
3 $I_1, \ldots, I_N \leftarrow \textsc{seqItp}(\varphi)$
4 $G_0 \leftarrow Init\ ;\ \forall 1 \leq i \leq N \cdot G_i \leftarrow F_i \wedge I_i$
5 return $[G_0, \ldots, G_N]$
```

Algorithm 2: IMCMKSAFE.

a candidate trace is made safe using IMCMKSAFE, if possible. The algorithm terminates when either a trace cannot be made safe, or when a closed trace is discovered.

IMCMKSAFE is shown in Alg. 2. The key insight is that a safe trace can be constructed by sequence interpolation. First, a BMC problem is solved to check for absence of a CEX. Second, a sequence interpolant is computed and is used to strengthen the current trace. Note that the sequence interpolant $Init, I_1, \ldots, I_N$ itself is a trace. Hence, correctness follows via closure of traces under conjunction.

The main advantage of IMC is that it integrates well with BMC, effectively turning incremental BMC into a complete Model Checking procedure. A main deficiency is that interpolants from one BMC check are not used to help the next one. An obvious improvement is to use the current trace to strengthen the BMC query at line 1 of IMCMKSAFE as follows:

$$\varphi \leftarrow Init(v_0) \wedge \bigwedge_{i=0}^{N-1} Tr(v_i, v_{i+1}) \wedge F_{i+1}(v_{i+1}) \wedge Bad(v_N) \tag{6}$$

This, however, is not effective in practice. The formulas F_i are typically large (as propositional formulas) and adding them significantly slows down BMC.

3.2 Property Directed Reachability

In this section, we give an overview of Property Directed Reachability (PDR/IC3) algorithm and its properties. Our presentation of PDR/IC3 is unorthodox, but

```
Input: Transition system $T = (Init, Tr, Bad)$
1 $F_0 \leftarrow Init$ ; $N \leftarrow 0$
2 repeat
3     $\boldsymbol{G} \leftarrow$ PdrMkSafe($[F_0, \ldots, F_N], Bad$)
4     if $\boldsymbol{G} = [\,]$ then return UNSAFE;
5     $\forall 0 \le i \le N \cdot F_i \leftarrow \boldsymbol{G}[i]$
6     $F_0, \ldots, F_N \leftarrow$ PdrPush($[F_0, \ldots, F_N]$)
      // $F_0, \ldots, F_N$ is a safe $\delta$-trace
7     if $\exists 0 \le i \le N \cdot F_i = \emptyset$ then return SAFE;
8     $N \leftarrow N + 1$ ; $F_N \leftarrow \emptyset$
9 until $\infty$;
```

Algorithm 3: PDR/IC3.

it highlights the parts necessary for understanding our new algorithm. For more details on PDR/IC3 the reader is referred to [4,10].

Like Imc, Pdr computes an inductive trace. Unlike Imc, Pdr does not use an unrolling of the transition system during the computation of the trace. Furthermore, the trace is kept monotone and clausal. To better explain the characteristics of the trace computed by Pdr, we introduce the notion of a δ-*trace*: A δ-*trace* is a sequence of formulas $[F_0, \ldots, F_N]$ such that the sequence $[G_0, \ldots, G_N]$, where $G_i = \bigwedge_{j=i}^{N} F_j$, is a monotone clausal trace. For a δ-trace $\boldsymbol{F}$, we write $\boldsymbol{F}_i^{\uparrow}$ for the ith element of the corresponding trace (i.e., G_i above). Note that a δ-trace $\boldsymbol{F}$ is closed if there exists an i such that $\boldsymbol{F}_i = \emptyset$.

Pdr is shown in Alg. 3. It maintains a loop invariant that $F_0, \ldots, F_N$ is a safe δ-trace (after line 6). Each iteration starts with a δ-trace that is safe except for the last element F_N. If possible, the trace is made safe via PdrMkSafe, otherwise the problem is decided UNSAFE. Then, the now safe δ-trace $F_0, \ldots, F_N$ is strengthened using PdrPush. PdrPush takes a δ-trace $\boldsymbol{F} = [F_0, \ldots, F_N]$ and returns a stronger *pushed* δ-trace $\boldsymbol{G} = [G_0, \ldots, G_N]$ defined as follows:

$$H_0 = F_0 \qquad H_i = F_i \cup \{c \in H_{i-1} \mid (H_{i-1}(u) \wedge Tr(u,v)) \to c(v)\} \tag{7}$$

$$G_N = H_N \qquad G_i = H_i \setminus H_{i+1} \text{ for } 0 \le i < N \tag{8}$$

If this closes the trace, the problem is decided SAFE. Otherwise, N is incremented and the loop is repeated.

PdrMkSafe takes a δ-trace $\boldsymbol{F} = [F_0, \ldots, F_N]$ that is safe except for F_N and makes it safe (by strengthening it) if possible, and, if not, returns an empty sequence. This is the main procedure of Pdr. We only give a high-level description of it here. Intuitively, PdrMkSafe does a backward search along the given trace $\boldsymbol{F}$, starting in some state $s_N \in Bad$ (recall, F_N is unsafe, so such s_N always exists). Then, a predecessor s_{N-1} is extracted from a model of $F_{N-1}(v) \wedge Tr(v,u) \wedge s_N(u)$. This is repeated until $Init$ is reached, or, for some i, $F_{i-1}(v) \wedge Tr(v,u) \wedge s_i(u)$ becomes UNSAT. In the latter case, $F_{i-1}(u) \wedge Tr(u,v) \to \neg s_i(u)$, and $\neg s_i$ can be conjoined (added as a clause) to F_i. PdrMkSafe improves this by a process called *inductive generalization*. Instead of adding $\neg s_i$ directly, it finds a sub-clause $c \to \neg s_i$ such that

$$Init \to c \qquad\qquad \boldsymbol{F}_i^{\uparrow}(u) \wedge c(u) \wedge Tr(u,v) \to c(v) \tag{9}$$

Such c is guaranteed to exist, in the worst case $\neg s_i$ is taken as c. Inductive generalization is often argued to be the most important element that contributes to the efficiency of Pdr. This process is continued until F_N becomes safe. An important property of PdrMkSafe is that it is guaranteed to find some safe strengthening of $\boldsymbol{F}$ if a strengthening exists.

Pdr offers many advantages, including incrementally (at each iteration only longer paths are explored) and locality of its SAT queries (all queries are over a single transition relation only). However, locality and the backward search strategy are also its Achilles' heel. There are many practical problems for which Imc's global and less directed search is superior.

4 Interpolating Property Directed Reachability

In this section, we introduce Avy, a Model Checking algorithm that, like Imc, uses BMC and sequence interpolants, and furthermore, like Pdr it uses backward search and inductive generalization. We first describe the basic building blocks of Avy, and then go into fine-grained details.

4.1 Basic Algortihm

Avy is shown in Alg. 4. Like Pdr it maintains a safe δ-trace $\boldsymbol{F} = [F_0, \ldots, F_N]$ and has the same high-level structure. However, the main steps for constructing the trace, making it safe (via AvyMkSafe) and maintaining δ-form (via AvyMkDelta), are done differently. We first give a high-level description of Avy and then of the two main functions.

Main loop. First, AvyMkSafe is used to check whether the current trace can be safely extended to the next bound. If possible, it returns a safe trace $\boldsymbol{G}$ that is stronger than $\boldsymbol{F}$. However, $\boldsymbol{G}$ is not necessarily a δ-trace. Second, AvyMkDelta strengthens (again) $\boldsymbol{G}$ and makes it a δ-trace. Finally, the algorithm continues as Pdr, using PdrPush to further strengthen the trace and check for convergence. In each iteration the trace can be incremented by an arbitrary step. But, for simplicity of presentation, assume that $step = 1$ unless stated otherwise. Note that the main loop maintains a safe δ-trace. Hence, in each iteration, the main loop of Pdr can be used instead, leading to an interleaved version of the two algorithms.

AvyMkSafe is presented in Alg. 5. It resembles ImcMkSafe, but with one key difference: it uses the existing trace to simplify both the BMC and interpolation problems (see line 1, where $\boldsymbol{F}_i^{\uparrow}$ is conjoined to the ith copy of the Tr). If the BMC formula φ is UNSAT, AvyMkSafe extracts the sequence interpolant and uses it to strengthen and extend the existing trace. Otherwise, φ is SAT and AvyMkSafe returns an empty trace.

There are multiple ways to partition the BMC formula φ for interpolation. To better understand the choice made in AvyMkSafe, consider the following example: $T = (\{x\}, x = 0, x' = x + 1, x \geq 6)$. T represents a simple counter that counts from 0, and the bad region is where the counter goes beyond 5. Let us assume that we have the following trace $[x = 0, x \leq 1, \top]$, and consider the BMC problem for bound 2, with the partitioning used by AvyMkSafe:

$$((x_0 = 0) \wedge (x_1 = x_0 + 1))_0 \wedge ((x_1 \leq 1) \wedge (x_2 = x_1 + 1))_1 \wedge (x_2 \geq 6)_2 \qquad (10)$$

An alternative way to partition the formula is to add the ith element of the trace to the $i - 1$ partition (for $i \geq 1$):

$$((x_0 = 0) \wedge (x_1 = x_0 + 1) \wedge (x_1 \leq 1))_0 \wedge (x_2 = x_1 + 1)_1 \wedge (x_2 \geq 6)_2 \qquad (11)$$

The choice of the partitioning influences the resulting sequence interpolant. In (10), the sequence interpolant contains only the parts that are needed to strengthen the existing trace. In (11), the interpolant is stronger than the trace (i.e., as if the trace was not added to the BMC formula).

In our example, in (10), since $x_1 \leq 1$ is strong enough, the suffix $((x_1 \leq 1) \wedge (x_2 = x_1 + 1))_1 \wedge (x_2 \geq 6)_2$ is UNSAT. By that we conclude that the first element of the sequence interpolant is $\top$. That is, F_1 in the trace needs no strengthening, which is evident in the resulting interpolant.

The example illustrates the advantage in choosing the partitioning used by Avy: the newly computed sequence interpolant takes into account the existing trace and only strengthens it as needed. This is part of the incrementality in Avy.

AvyMkDelta is shown in Alg. 6. We first describe the intuition, then the mechanics. AvyMkDelta converts a safe trace $\boldsymbol{G} = [G_0, \ldots, G_N]$ into a monotone and clausal trace $\boldsymbol{F} = [F_0, \ldots, F_N]$. Note that the result of AvyMkSafe is safe but neither monotone nor clausal. One alternative to making a trace $[G_0, \ldots, G_N]$ monotone is to replace each element G_i by a disjunction of its predecessors $\{G_j\}_{j<i}$, i.e., by letting $F_i = \bigvee_{j<i} G_j$. But this is inefficient because the resulting formulas are too large.

Another alternative is to use interpolation. For example, let $[\mathit{Init}, G_1, G_2]$ be a safe but non-monotone and non-clausal trace. To make it monotone, we need $\mathit{Init} \to G_1$ and $G_1 \to G_2$. For the first implication, create the following problem

$$A = \mathit{Init}(v) \vee (\mathit{Init}(u) \wedge \mathit{Tr}(u, v)) \qquad B = \neg \mathit{Init}(v) \wedge \neg G_1(v) \qquad (12)$$

From the definition of a trace, $A \wedge B$ is unsatisfiable. Let F_1 be a corresponding interpolant. By construction, $\mathit{Init} \to F_1$ and $\mathit{Init}(u) \wedge \mathit{Tr}(u, v) \to F_1(v)$. For the second implication, we compute an interpolant F_2 between $A = F_1(v) \vee (F_1(u) \wedge \mathit{Tr}(u, v))$ and $B = \neg(F_1(v) \vee G_2(v))$. F_2 satisfies: $F_1 \to F_2$ and $F_1(v) \wedge \mathit{Tr}(v, v') \to F_2(v')$. Hence, the trace $[\mathit{Init}, F_1, F_2]$ is safe and monotone.

However, in addition to monotonicity, we require that the trace is a clausal δ-trace. Transforming an arbitrary propositional formula into CNF without adding new variables is expensive. One possibility is to generate interpolants in CNF by a CNF-producing interpolation procedure (e.g., [17]). While [17] is efficient it does not generate a δ-trace.

```
Input: Transition system $T = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$
1 $F_0 \leftarrow \mathit{Init}\,; N \leftarrow 0$
2 repeat
3     $\boldsymbol{G} \leftarrow$ AvyMkSafe($[F_0, \ldots, F_N], \mathit{Bad}$)
4     if $\boldsymbol{G} = [\,]$ then return UNSAFE;
5     $F_0, \ldots, F_N \leftarrow$ AvyMkDelta($\boldsymbol{G}$)
6     $F_0, \ldots, F_N \leftarrow$ PdrPush($[F_0, \ldots, F_N]$)
      // $F_0, \ldots, F_N$ is a safe $\delta$-trace
7     if $\exists 0 \leq i \leq N \cdot F_i = \emptyset$ then return SAFE;
8     pick $\mathit{step} \geq 1$
9     $\forall N \leq i < N + \mathit{step} \cdot F_i \leftarrow \emptyset$
10    $N \leftarrow N + \mathit{step}$
11 until $\infty$;
```
Algorithm 4: Avy (simplified).

```
Input: Transition system $T = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$
Input: A $\delta$-trace $\boldsymbol{F} = [F_0, \ldots, F_N]$
1 $\varphi \leftarrow \bigwedge_{i=0}^{N-1} \left(\boldsymbol{F}_i^{\uparrow}(v_i) \land \mathit{Tr}(v_i, v_{i+1})\right)_i \land (\boldsymbol{F}_N^{\uparrow}(v_N) \land \mathit{Bad}(v_N))_N$
2 if isSat($\varphi$) then return $[\,]$;
3 $I_1, \ldots, I_N \leftarrow$ seqItp($\varphi$)
4 $G_0 \leftarrow \mathit{Init}\,; \forall 1 \leq i \leq N \cdot G_i \leftarrow \boldsymbol{F}_i^{\uparrow} \land I_i$
5 return $[G_0, \ldots, G_N]$
```
Algorithm 5: AvyMkSafe.

Instead, we have chosen to re-use Pdr's PdrMkSafe that already maintains a δ-trace. Our unorthodox use of PdrMkSafe is guided towards our purpose. We establish the correctness of this method at the end of the section.

As before, consider a non-monotone non-clausal trace $[\mathit{Init} = G_0, G_1, G_2]$. Recall that PdrMkSafe takes a δ-trace and returns a strengthened safe δ-trace w.r.t. a given property. For the first element of the trace, we define $[\mathit{Init}, \top]$ as the input δ-trace. Then, PdrMkSafe is used to transform this δ-trace into a safe δ-trace w.r.t. the property $\mathit{Init} \lor G_1$. The result of PdrMkSafe is therefore a safe δ-trace $[\mathit{Init}, F_1]$ s.t. $\mathit{Init} \rightarrow \boldsymbol{F}_1^{\uparrow}$ and $\mathit{Init}(u) \land \mathit{Tr}(u, v) \rightarrow \boldsymbol{F}_1^{\uparrow}(v)$. For the second element G_2, the δ-trace $[\mathit{Init}, F_1, \top]$ is used. Now, PdrMkSafe is used to transform w.r.t. the property $\boldsymbol{F}_1^{\uparrow} \lor G_2$. The result is again, a safe δ-trace $[\mathit{Init}, F_1, F_2]$ s.t. the previous holds and in addition, $\boldsymbol{F}_1^{\uparrow} \rightarrow \boldsymbol{F}_2^{\uparrow}$ and $\boldsymbol{F}_1^{\uparrow}(u) \land \mathit{Tr}(u, v) \rightarrow \boldsymbol{F}_2^{\uparrow}(v)$. The general version of this algorithm is shown in Alg. 6.

We conclude with an outline of the correctness argument. To show correctness, it is enough to show that (a) AvyMkSafe always returns a safe trace if possible, and (b) AvyMkDelta returns a safe δ-trace given a safe trace. The rest of the proof (both for soundness and completeness) is the same as for Pdr. Part (a) is an immediate consequence of sequence interpolation property, and we do not expand on it further. To show (b), we need to show that (i) the calls to

```
    Input: Transition system $T = (Init, Tr, Bad)$
    Input: A safe trace $\boldsymbol{G} = [G_0, \ldots, G_N]$
    Output: A safe $\delta$-trace $\boldsymbol{F} = [F_0 \ldots, F_N]$
1   $F_0 \leftarrow Init$
2   $[\_, F_1] \leftarrow$ PdrMkSafe($[Init, \top], \neg(Init \vee G_1)$)
3   for $i \leftarrow 2$ to $N$ do
4       $[\_, \_, F_i] \leftarrow$ PdrMkSafe($[Init, F_{i-1}, \top], \neg(F_{i-1} \vee G_i)$)
5   end
```

Algorithm 6: AvyMkDelta.

PdrMkSafe always return a safe δ-trace, and (ii) δ-traces can be concatenated together. Part (ii) is an immediate consequence of the δ-trace property:

Lemma 1. *If $\boldsymbol{F} = [Init, F_1, \ldots, F_N]$ and $\boldsymbol{G} = [Init, F_N, G_2]$ are safe δ-traces, then so is $[Init, F_1, \ldots, F_N, G_2]$.*

To establish (i), we only need to show that the input to PdrMkSafe can be made safe. For the call at line 2 of AvyMkDelta, by the trace property of $\boldsymbol{G}$, $Init(u) \wedge Tr(u, v) \to G_1(v)$. For the call at line 4, we show by induction that $(F_{i-1}(u) \wedge Tr(u, v)) \to (F_{i-1}(v) \vee G_i(v))$. The base case is $i = 2$. We know that $F_1 \to (Init \vee G_1)$ (the call at line 2). Since both $[G_0, G_1, G_2]$ and $[Init, F_1]$ are traces, we have: $(G_1(u) \wedge Tr(u, v)) \to G_2(v))$ and $(Init(u) \wedge Tr(u, v)) \to F_1(v)$. By these three facts we get $(F_1(u) \wedge Tr(u, v)) \to (F_1(v) \vee G_2(v))$. The inductive case is similar. Using $(F_{i-1}(u) \wedge Tr(u, v)) \to (F_{i-1}(v) \vee G_i(v))$, we can conclude that each call at line 4 does not change F_{i-1} and thus Lemma 1 is applicable.

Theorem 2. Avy *is sound and complete for $step = 1$.*

When $step > 1$, AvyMkSafe is not guaranteed to return a safe trace. While the last frame is safe, the intermediate ones might not be. One way around this is to require that Tr gets trapped in the Bad region.

Definition 1 (Stuck-On-Error). *A transition system $T = (Init, Tr, Bad)$ is stuck-on-error iff $\forall s \in Bad \cdot \exists t \in Bad \cdot Tr(s, t)$.*

Note that stuck-on-error can be enforced for any Tr by adding a self-loop on all Bad states. The rest of the proof remains unchanged.

Theorem 3. Avy *is sound and complete for $step > 1$ for any transition system T that satisfies stuck-on-error property of Def. 1.*

4.2 The Whole Picture

In the previous section, we gave a simplified description of Avy. Here, we describe some of its key features. The complete algorithm is shown in Alg. 7. The biggest change is that this version combines all the steps into a single function. In the rest of the section, we explain some features in detail.

Global δ-trace. Unlike the simplified presentation before, this version maintains a single global δ-trace. At every iteration, $\boldsymbol{F}$ is used incrementally by adding missing clauses. This is evident at lines 6–8. Note that both at line 6 and at line 8, the δ-trace that is given to PdrMkSafe already has clauses that were learned in previous iterations. Hence, when transforming the newly generated interpolant to CNF, only clauses that are missing are added to $\boldsymbol{F}$. This eliminates an expensive clause re-learning of the simplified version of the algorithm.

Guided Proofs. The upside of relying on interpolation is that Avy does not interfere with the SAT-solver during the BMC step. The downside is that, compared to Pdr, there is very little control on the quality of the generated lemmas. A solution we adopt is to "guide" the SAT-solver that is producing the proof for interpolation. This is done by asking the solver to produce Minimal Unsatisfiable Subset (MUS) that excludes as many clauses from Tr and includes as many clauses from $\boldsymbol{F}$ as possible. The choice of a MUS affects the quality of the generated interpolants, and the choice of MUS algorithm affects the efficiency. In our implementation, we use a basic MUS algorithm (cf. [14]), and the MUS strategies described next.

We have tried two strategies for guiding the proof. First, called **min-core**, simply computes the MUS, letting the MUS algorithm pick which clauses to select. While this strategy is very fine grained, it was not effective in practice. It did cause an order of magnitude improvement in one example, but degraded performance overall.

The second strategy, called **min-suffix**, attempts to find a MUS that completely contains a suffix of the BMC problem. That is, it looks for the largest k such that $(\bigwedge_{i=k}^{N-1} \boldsymbol{F}_i^{\uparrow}(v_i) \wedge Tr(v_i, v_{i+1})) \wedge \boldsymbol{F}_N^{\uparrow}(v_N) \wedge Bad(v_N)$ is unsatisfiable.

To illustrate, consider the example from the previous section (reproduced here for convenience):

$$((x_0 = 0) \wedge (x_1 = x_0 + 1))_0 \wedge ((x_1 \leq 1) \wedge (x_2 = x_1 + 1))_1 \wedge (x_2 \geq 6)_2 \qquad (13)$$

Recall, $x \leq 1$ is sufficient and, therefore, **min-suffix** reduces it to:

$$(\top)_0 \wedge ((x_0 \leq 1) \wedge (x_1 = x_0 + 1))_1 \wedge (x_1 \geq 6)_2 \qquad (14)$$

The immediate benefits of **min-suffix** are: (a) the solved BMC formula is simpler (shorter bound); (b) the extracted sequence interpolant is smaller and, therefore, less interpolants need to be transformed to monotone clausal form; and (c) the proof is guided towards the important facts (e.g., to $x \leq 1$ in the case above). This makes generalization more effective.

Shallow Push. At each iteration of trace strengthening, new clauses are added to the global trace $\boldsymbol{F}$. Therefore, it is possible to push the clauses forward after adding them (line 9) as they might be useful for the next iteration. Note that this is very different from the simplified version of the algorithm. There, the pushing-phase happens only after all of the strengthening. In practice, we push more conservatively, to which we refer as *shallow push.* During shallow push, clauses are only pushed starting from the ith location (where clauses were just added). This way, in the next iteration, when PdrMkSafe is applied, it may need to find less clauses (or even none at all).

```
Input: Transition system $T = (Init, Tr, Bad)$
Data: A $\delta$-trace $\boldsymbol{F} = [F_0, \ldots, F_N]$
1  $F_0 \leftarrow Init$ ; $N \leftarrow 0$
2  repeat
3      $\varphi \leftarrow \bigwedge_{i=0}^{N-1} \left(\boldsymbol{F}_i^{\uparrow}(v_i) \wedge Tr(v_i, v_{i+1})\right)_i \wedge (\boldsymbol{F}_N^{\uparrow}(v_N) \wedge Bad(v_N))_N$
4      if isSat($\varphi$) then return UNSAFE;
5      $I_1, \ldots, I_N \leftarrow$ seqItp($\varphi$)
6      $[\_, F_1] \leftarrow$ PdrMkSafe($[Init, \boldsymbol{F}_1^{\uparrow}], \neg(Init \vee I_1)$)
7      for $i \leftarrow 2$ to $N$ do
8          $[\_, \_, F_i] \leftarrow$ PdrMkSafe($[Init, \boldsymbol{F}_{i-1}^{\uparrow}, \boldsymbol{F}_i^{\uparrow}], \neg(\boldsymbol{F}_{i-1}^{\uparrow} \vee I_i)$)
9          $F_0, \ldots, F_N \leftarrow$ PdrPush($[F_0, \ldots, F_N]$)
10     end
       // $F_0, \ldots, F_N$ is a safe $\delta$-trace
11     if $\exists 0 \leq i \leq N \cdot F_i = \emptyset$ then return SAFE;
12     pick $step \geq 1$
13     $\forall N \leq i < N + step \cdot F_i \leftarrow \emptyset$
14     $N \leftarrow N + step$
15 until $\infty$;
```

Algorithm 7: Avy.

Table 1: Summary of solved instances on HWMCC'12 and HWMCC'13. CNF-ITP appears with (*) since we were not able to run it on the entire HWMCC'13 benchmark due to technical issues.

Status	Avy	Pdr	Itp	CNF-ITP	Virtual Best
SAFE	76	72	62	59(*)	112
UNSAFE	24	15	26	25(*)	29

5 Experiments

We have implemented Avy[4] using C++ on top of ABC [5] – a well known open-source verification framework. We have compared it on HWMCC'12 and HWMCC'13 benchmark suites against Pdr, McMillan's Interpolation algorithm (Itp) [12] as implemented in ABC, and CNF-ITP [17]. Note that Itp is slightly different from Imc described in Section 3.1. While an efficient implementation of Imc was not available, prior experiments indicate that Itp outperforms Imc on HWMCC benchmarks [6]. All experiments were performed on Intel E5-2697V2 running at 2.7GHz and with 256GB of RAM with a 900 seconds timeout.

We have joined HWMCC'12 and HWMCC'13 together into a set of benchmarks, excluding Beem [5] test cases as we put emphasis on the industrial section of the benchmark (which includes 328 test cases).

The results are summarized in Table 1. Avy dominates the benchmark in number of solved instances. In particular, on the Intel set, Avy and CNF-

[4] Available at `http://www.cs.technion.ac.il/~yvizel/avy.html`.
[5] `http://paradise.fi.muni.cz/beem`.

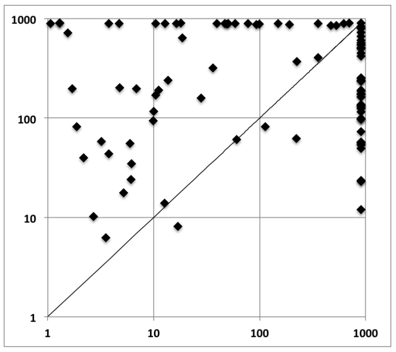
Table 2: Detailed experimental results. D represents the depth of convergence, $\sharp$ Clauses - the number of clauses in the proof, and *Time* is the runtime in seconds. (*) Note that CNF-ITP failed to run on the OSKI cases due to technical issues.

Test	**Status**	**ITP**		**CNF-ITP**			**PDR**			**Avy**		
		D	Time[s]	D	$\sharp$ Clauses	Time[s]	D	$\sharp$ Clauses	Time[s]	D	$\sharp$ Clauses	Time[s]
6s102	T	53	TO	46	16,350	111	13	2966	222.22	23	162	61.92
6s121	T	342	TO	42	2,907	13.2	17	–	TO	49	1,713	499.14
6s130	T	14	18.66	18	93,600	856	7	–	TO	9	2,669	114.7
6s144	T	35	TO	23	–	TO	9	–	TO	22	371	449.53
6s159	T	63	11.5	10	280	0.3	45	114	2.7	36	19	10.2
6s189	T	37	TO	23	–	TO	8	–	TO	26	384	793.15
6s194	T	70	TO	80	–	TO	38	4,763	93.32	50	–	TO
6s205b16	T	61	213.01	35	–	TO	43	–	TO	10	–	TO
6s206rb025	T	7	2.51	6	24	2.5	4	8	0.22	4	8	8.28
6s207rb16	F	9	2.52	10	–	TO	5	–	TO	8	–	22.94
6s282b15	T	33	13.38	33	49,025	65	19	1,576	9.99	25	697	116.59
6s288r	T	83	TO	40	3,998	155	19	236	10.38	21	106	170.49
6s131	T	13	19.18	20	–	TO	6	–	TO	8	2,626	96.88
6s162	F	73	217.72	73	–	TO	13	–	TO	72	–	173.63
6s38	T	23	TO	24	4,508	558	9	–	TO	12	1,193	130.15
6s407rb296	T	18	TO	9	–	TO	9	–	TO	12	238	173.18
6s408rb191	T	37	TO	16	33,116	228	6	883	0.97	8	644	199.94
6s8	T	43	TO	38	–	TO	26	–	TO	35	2,021	829.12
6s9	T	14	30.56	10	–	TO	9	–	TO	8	2,727	96.85
intel011	T	72	TO	20	–	TO	27	–	TO	52	572	233.94
intel015	T	72	TO	21	–	TO	51	–	TO	60	726	124.29
intel018	T	78	TO	16	–	TO	50	–	TO	60	328	56.6
intel020	T	90	TO	15	3,975	48	33	–	TO	46	370	56.28
intel021	T	92	TO	18	5,958	115	33	–	TO	52	365	99.62
intel022	T	84	TO	21	–	TO	27	–	TO	38	405	73.18
intel023	T	96	TO	32	9,312	606	30	–	TO	50	243	57.09
intel024	T	96	TO	15	4,395	78	23	–	TO	38	194	23.43
intel025	T	60	TO	17	–	TO	23	–	TO	42	1,204	421.07
intel029	T	84	TO	16	–	TO	47	–	TO	54	230	53.31
intel034	T	86	TO	16	1,344	119	55	–	TO	72	232	603.85
oski1rub03	T	9	4.02	–(*)	–(*)	–(*)	8	169	12.71	6	43	13.96
oski1rub04	F	13	28.46	–(*)	–(*)	–(*)	14	–	112.42	12	–	81.89
oski1rub07	T	4	1.22	–(*)	–(*)	–(*)	7	144	3.51	2	140	6.22

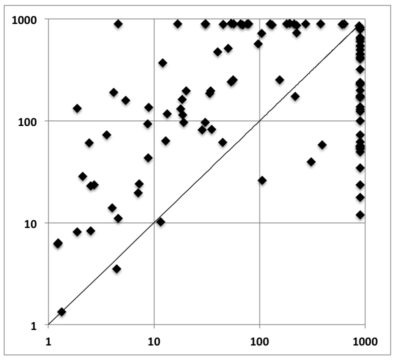
ITP are the only techniques able to solve safe instances, though Avy solves considerably more instances than CNF-ITP. Inspecting the entire set of solved instances, the instances solved by Avy and Pdr are significantly different. The "Virtual Best" column shows the result of a solver that runs all 3 techniques and takes the best result. It shows that Avy is complimentary to Pdr. Together, they solve at least a third more benchmarks than either one in isolation.

More details are shown in Table 2. There are two important parameters to notice: the depth at which a proof (fixpoint) is found and the number of clauses in the proof. On the cases where both Pdr and Avy reach to a fixpoint, the number of clauses in the proof Avy finds is smaller than those in the proof found by Pdr, even in the cases where Pdr converges at a lower depth.

The run-time results for the entire benchmark are shown in Fig. 1. In all plots, Avy is represented by the y-axis. While whenever Avy solves a problem that is solved by another method, it is slower, it solves a large number of problems



(a) PDR vs. AVY: All.



(b) ITP vs. AVY: All.

Fig. 1: Runtime comparison between AVY (y-axis) and PDR and ITP.

not solved by other techniques. We believe that the performance issues are in part due to our implementation of interpolation and lack of support for the combination of incremental SAT-solving and interpolation.

We have also evaluated the effect of specific techniques used by AVY and found all of them to be important. AVY is not competitive if any of them are disabled. In particular, maintaining the global δ-trace and guiding the proof towards minimal unsatisfiable suffix are critical to performance. In addition, 3 test cases were only solved with the **min-core** option.

6 Conclusion

We introduce AVY, a new SAT-based model checking algorithm. Like IMC and PDR, AVY constructs a safe inductive invariant to show the validity of a property. It uses BMC-unrolling with sequence interpolants to construct an initial candidate invariant (similar to IMC), but then uses local backward search and inductive generalization to keep the candidate invariant in a compact clausal form. AVY combines the advantages of both IMC and PDR. Our experiments show that AVY is a very capable algorithm that can solve a considerable number of test cases that are not solvable by neither PDR nor ITP and CNF-ITP.

As future directions, we would like to experiment with other methods that can keep the trace in compact clausal form (e.g., using the approach from [17]). In addition, we believe that the concepts that were introduced in this paper extends beyond finite state systems and can be applied in the context of software model checking.

References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig Interpretation. In A. Miné and D. Schmidt, editors, *SAS*, volume 7460 of *Lecture Notes in Computer Science*, pages 300–316. Springer, 2012.

2. A. Albarghouthi, A. Gurfinkel, Y. Li, S. Chaki, and M. Chechik. UFO: Verification with Interpolants and Abstract Interpretation - (Competition Contribution). In *TACAS*, pages 637–640, 2013.
3. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
4. A. R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, pages 70–87, 2011.
5. R. K. Brayton and A. Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010.
6. G. Cabodi, S. Nocco, and S. Quer. Interpolation sequences revisited. In *DATE*, pages 316–322. IEEE, 2011.
7. H. Chockler, A. Ivrii, and A. Matsliah. Computing interpolants without proofs. In A. Biere, A. Nahir, and T. E. J. Vos, editors, *Haifa Verification Conference*, volume 7857 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 2012.
8. W. Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. of Symbolic Logic*, 22(3):269–285, 1957.
9. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In G. Barthe and M. V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010.
10. N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In P. Bjesse and A. Slobodová, editors, *FMCAD*, pages 125–134. FMCAD Inc., 2011.
11. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.
12. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13, 2003.
13. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.
14. A. Nadel. Boosting minimal unsatisfiable core extraction. In R. Bloem and N. Sharygina, editors, *FMCAD*, pages 221–229. IEEE, 2010.
15. Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In *FMCAD*, pages 1–8. IEEE, 2009.
16. Y. Vizel, O. Grumberg, and S. Shoham. Intertwined forward-backward reachability analysis using interpolants. In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 308–323. Springer, 2013.
17. Y. Vizel, V. Ryvchin, and A. Nadel. Efficient generation of small interpolants in CNF. In *CAV*, pages 330–346, 2013.