Searching for i-Good Lemmas to Accelerate Safety Model Checking*

Yechuan Xia[1], Anna Becchi[2], Alessandro Cimatti[2], Alberto Griggio[2], Jianwen Li[1], and Geguang Pu[1,3]

[1] East China Normal University, Shanghai, China
xiaozi465@gmail.com, {jwli,ggpu}@sei.ecnu.edu.cn
[2] Fondazione Bruno Kessler, Trento, Italy
{abecchi,cimatti,griggio}@fbk,eu
[3] Shanghai Trusted Industrial Control Platform Co., Ltd, Shanghai, China

Abstract. IC3/PDR and its variants have been the prominent approaches to safety model checking in recent years. Compared to the previous model-checking algorithms like BMC (Bounded Model Checking) and IMC (Interpolation Model Checking), IC3/PDR is attractive due to its completeness (vs. BMC) and scalability (vs. IMC). IC3/PDR maintains an over-approximate state sequence for proving the correctness. Although the sequence refinement methodology is known to be crucial for performance, the literature lacks a systematic analysis of the problem. We propose an approach based on the definition of *i-good lemmas*, and the introduction of two kinds of heuristics, i.e., branching and refer-skipping, to steer the search towards the construction of i-good lemmas. The approach is applicable to IC3 and its variant CAR (Complementary Approximate Reachability), and it is very easy to integrate within existing systems. We implemented the heuristics into two open-source model checkers, IC3Ref and SimpleCAR, as well as into the mature nuXmv platform, and carried out an extensive experimental evaluation on HWMCC benchmarks. The results show that the proposed heuristics can effectively compute more i-good lemmas, and thus improve the performance of all the above checkers.

1 Introduction

Safety model checking is a fundamental problem in verification. The goal is to prove that all the reachable states of the transition system $\langle I, T\rangle$ satisfy a property P. The field has been dominated by SAT-based techniques since the introduction of Bounded Model Checking (BMC) [9]. The first wave of SAT-based model-checking algorithms, including BMC, k-induction [31] and Interpolation-based Model Checking [25] have been superseded by the research deriving from the seminal work of Bradley [11]. The IC3 algorithm maintains an over-approximate state sequence for proving the correctness; it avoids unrolling the transition relation by localizing reasoning to *frames*, used to incrementally build an inductive invariant by discovering inductive clauses.

* Jianwen Li and Geguang Pu are corresponding authors

IC3 (also known as PDR [17]) has spawned several variants, including those that attempt to combine forward and backward search [29]. Particularly relevant in this paper is CAR (Complementary Approximate Reachability), which combines the forward overapproximation with a backward underapproximation [23].

It has been noted that different ways to refine the over-approximating sequence can impact the performance of the algorithm. For example, [21] attempts to discover *good* lemmas, that can be "pushed to the top" since they are inductive. In this paper, we propose an alternative way to drive the refinement of the over-approximating sequence. We identify *i-good lemmas*, i.e. lemmas that are inductive with respect to the i-th overapproximating level. The intuition is that such i-good lemmas are useful in the search since they are fundamental to reach a fix point in the safe case. In order to guide the search towards the discovery of i-good lemmas, we propose a heuristic approach based on two key insights, i.e., branching and refer-skipping. First, with branching we try to control the way the SAT solver extracts unsatisfiable cores by privileging variables occurring in i-good lemmas. Second, we control lemma generalization by avoiding dropping literals occurring in a subsuming lemma in the previous layer (refer-skipping).

The proposed approach is applicable both to IC3/PDR and CAR, and it is very simple to implement. Yet, it appears to be quite effective in practice. We implemented the i-good lemma heuristics in two open-source implementations of IC3 and CAR, and also in the mature, state-of-the-art IC3 implementation available inside the nuXmv model checker [12], and we carried out an extensive experimental evaluation on Hardware Model Checking Competition (HWMCC) benchmarks. Analysis of the results suggests that increasing the ratio of i-good lemmas leads to an increase in performance, and the heuristics appear to be quite effective in driving the search towards i-good lemmas. In terms of performance, this results in significant improvements for all the tools when equipped with the proposed approach.

This paper is structured as follows. In Section 2 we present the problem and the IC3/PDR and CAR algorithms. In Section 3 we present the intuition underlying i-good lemmas and the algorithms to find them. In Section 4 we overview the related work. In Section 5 we present the experimental evaluation. In Section 6 we draw some conclusions and present directions for future work.

2 Preliminaries

2.1 Boolean Transition System

A Boolean transition system *Sys* is a tuple $\langle X, Y, I, T\rangle$, where X and X' denote the set of state variables in the present state and the next state, respectively, and Y denotes the set of input variables. The state space of *Sys* is the set of possible assignments to X. $I(X)$ is a Boolean formula corresponding to the set of initial states, and $T(X, Y, X')$ is a Boolean formula representing the transition relation. State s_2 is a successor of state s_1 with input y iff $s_1 \wedge y \wedge s_2' \models T$, which is also denoted by $(s_1, y, s_2) \in T$. In the following, we will also write $(s_1, s_2) \in T$

meaning that $(s_1, y, s_2) \in T$ for some assignment y to the input variables. A *path* of length k is a finite state sequence $s_1, s_2, \ldots, s_k$, where $(s_i, s_{i+1}) \in T$ holds for $(1 \leq i \leq k-1)$. A state t is reachable from s in k steps if there is a path of length k from s to t. Let S be a set of states in Sys. We overload T and denote the set of successors of states in S as $T(S) = \{t \mid (s,t) \in T, s \in S\}$. Conversely, we define the set of predecessors of states in S as $T^{-1}(S) = \{s \mid (s,t) \in T, t \in S\}$. Recursively, we define $T^0(S) = S$ and $T^{i+1}(S) = T(T^i(S))$ where $i \geq 0$; the notation $T^{-i}(S)$ is defined analogously. In short, $T^i(S)$ denotes the states that are reachable from S in i steps, and $T^{-i}(S)$ denotes the states that can reach S in i steps.

2.2 Safety Checking and Reachability Analysis

Given a transition system $Sys = \langle X, Y, I, T \rangle$ and a safety property P, which is a Boolean formula over X, a model checker either proves that P holds for any state reachable from an initial state in I, or disproves P by producing a *counterexample*. In the former case, we say that the system is *safe*, while in the latter case, it is *unsafe*. A *counterexample* is a finite path from an initial state s to a state t violating P, i.e., $t \in \neg P$, and such a state is called a *bad* state. In symbolic model checking, safety checking is reduced to symbolic reachability analysis. Reachability analysis can be performed in a forward or backward search. Forward search starts from initial states I and searches for bad states by computing $T^i(I)$ with increasing values of i, while backward search begins with states in $\neg P$ and searches for initial states by computing $T^{-i}(\neg P)$ with increasing values of i. Table 1 gives the corresponding formal definitions.

Table 1. Exact reachability analysis.

	Forward	Backward
Base	$F_0 = I$	$B_0 = \neg P$
Induction	$F_{i+1} = T(F_i)$	$B_{i+1} = T^{-1}(B_i)$
Safe Check	$F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$	$B_{i+1} \subseteq \bigcup_{0 \leq j \leq i} B_j$
Unsafe Check	$F_i \cap \neg P \neq \emptyset$	$B_i \cap I \neq \emptyset$

For forward search, F_i denotes the set of states that are reachable from I within i steps, which is computed by iteratively applying T. At each iteration, we first compute a new F_i, and then perform safe checking and unsafe checking. If the safe/unsafe checking hits, the search terminates. Intuitively, unsafe checking $F_i \cap \neg P \neq \emptyset$ indicates some bad states are within F_i and safe checking $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$ indicates that all reachable states from I have been checked and none of them violate P. For backward search, B_i is the set of states that can reach $\neg P$ in i steps, and the search procedure is analogous to the forward one.

Notations. A *literal* is an atomic variable or its negation. If l is a literal, we denote its corresponding variable with $var(l)$. A *cube* (resp. *clause*) is a conjunction (resp. disjunction) of literals. The negation of a clause is a cube and vice

versa. A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses. For simplicity, we also treat a CNF formula ϕ as a set of clauses and make no difference between the formula and its set representation. Similarly, a cube or a clause c can be treated as a set of literals or a Boolean formula, depending on the context.

We say a CNF formula ϕ is satisfiable if there exists an assignment of its Boolean variables, called a *model*, that makes ϕ true; otherwise, ϕ is unsatisfiable. A SAT solver is a tool that can decide the satisfiability of a CNF formula ϕ. In addition to providing a yes/no answer, modern SAT solvers can also produce *models* for satisfiable formulas, and *unsatisfiable cores* (UC), i.e. a reason for unsatisfiability, for unsatisfiable ones. More precisely, in the following we shall assume to have a SAT solver that supports the following API (which is standard in state-of-the-art SAT solvers based on the CDCL algorithm [24]):

- **is_SAT**$(\phi, \mathcal{A})$ checks the satisfiability of ϕ under the given assumptions $\mathcal{A}$, which is a list of literals. This is logically equivalent to checking the satisfiability of $\phi \wedge \bigwedge \mathcal{A}$, but is typically more efficient;
- **get_UC**() retrieves an UC of the assumption literals of the previous SAT call when the formula $\phi \wedge \bigwedge \mathcal{A}$ is unsatisfiable. That is, the result is a set $uc \subseteq \mathcal{A}$ such that $\phi \wedge \bigwedge uc$ is unsatisfiable;
- **get_model**() retrieves the model of the formula $\phi \wedge \bigwedge \mathcal{A}$ of the previous SAT call, if the formula is satisfiable.

2.3 Overview of IC3 and CAR

IC3 is a SAT-based and complete safety model checking algorithm proposed in [11], which only needs to unroll the system at most once. PDR [17] is a re-implementation of IC3 which optimizes the original version in different aspects. To prove the correctness of a given system $Sys = \langle X, Y, I, T \rangle$ w.r.t. the safety property P, IC3/PDR maintains a monotone over-approximate state sequence O such that (1) $O_0 = I$ and (2) $O_{i+1} \supseteq O_i \cup T(O_i)$ for $i \geq 0$. From the perspective of reachability analysis, IC3 performs as shown in the left part of Table 2. Since O is monotone, the states search can converge as soon as $O_{i+1} = O_i$ holds for some $i \geq 0$. Otherwise, a state path (counterexample) starting from I to some state in $\neg P$ can be detected ($T^{-i}(\neg P) \cap I \neq \emptyset$).

Table 2. A high-level description of IC3 (left) and (Forward) CAR (right).

	Over-approximate	Under-approximate
Base	$O_0 = I$	-
Induction	$O_{i+1} \supseteq O_i \cup T(O_i)$	-
Safe Check	$\exists i \cdot O_{i+1} = O_i$	-
Unsafe Check	-	$\exists i \cdot T^{-i}(\neg P) \cap I \neq \emptyset$

	Over-approximate	Under-approximate
Base	$O_0 = I$	$U_0 = \neg P$
Induction	$O_{i+1} \supseteq T(O_i)$	$U_{i+1} \subseteq T^{-1}(U_i)$
Safe Check	$\exists i \cdot O_{i+1} \subseteq \bigcup_{0 \leq j \leq i} O_j$	-
Unsafe Check	-	$\exists i \cdot U_i \cap I \neq \emptyset$

CAR [23] is a recently proposed algorithm, which can be considered as a general version of IC3. The main points CAR differs from IC3 are as follows:

Algorithm 1 Overview of IC3

```
1:  procedure IC3(I, T, P)
2:      if is_SAT(I ∧ ¬P) then                          // unsafe check of initial state
3:          return unsafe
4:      O_0 := I, k := 1, O_k := ⊤
5:      while true do
6:          while is_SAT(O_k ∧ ¬P) do
7:              s := get_model()                        // s ⊨ O_k ∧ ¬P
8:              if UnsafeCheck(s, k − 1) then
9:                  return unsafe                       // counterexample found
10:         k := k + 1, O_k := ⊤
11:         if SafeCheck(k) then
12:             return safe                             // property proved
13:
14: function UnsafeCheck(s, i)
15:     while is_SAT(O_i ∧ ¬s ∧ T, s′) do
16:         if i = 0 then
17:             return true
18:         t :=get_predecessor(s, i)                   // (t, s) ∈ T, see Alg. 4
19:         if UnsafeCheck(t, i − 1) then
20:             return true
21:     c := generalize({l|l′ ∈ get_UC()}, i)           // c ⊆ s, see Alg. 3
22:     O_j := O_j ∩ ¬c, 1 ≤ j ≤ i + 1
23:     return false
24:
25: function SafeCheck(k)
26:     propagation(k)                                  // see Alg. 4
27:     i := 0
28:     while i < k do
29:         if O_i = O_{i+1} then
30:             return true
31:     return false
```

- The over-approximate state sequence O in CAR is not necessarily monotone. Therefore, CAR has to apply the standard invariant-checking approach, i.e., finding a position $i \geq 0$ such that $O_{i+1} \subseteq \bigcup_{0 \leq j \leq i} O_j$ holds, as shown in the right part of Table 2.
- Besides the O sequence, CAR also maintains an under-approximate state sequence U that stores reachable (real) states from $\neg P$, see Table 2. The motivation to introduce the U sequence is to re-use the intermediate states that are computed during proving. Although it is straightforward for IC3 to introduce such a sequence, the effect on the performance remains unknown.
- CAR can be performed in both forward, i.e., proving from I while searching states back from $\neg P$, and backward, i.e., proving back from $\neg P$ while searching states from I. Although Backward CAR is not good at proving, it is advantageous in finding bugs, i.e., checking unsafety [22, 16]. Relevant work on reverse IC3/PDR [28], which corresponds to Backward CAR, has been studied but the results did not clearly support its advantage on bug-finding.

An overview of IC3 and (forward) CAR is shown in Alg. 1 and Alg. 2 respectively. At a high level, both algorithms have a similar structure, consisting of an alternation of two phases: unsafe check and safe check. The unsafe check (line 14 of Alg. 1, line 14 of Alg. 2) tries to find a state sequence that is a path between I and $\neg P$; if such a sequence can be found, then it is a counterexample witnessing the violation of P; otherwise, the O_i are strengthened with additional clauses

Algorithm 2 Overview of CAR

```
 1: procedure CAR_forward(I, T, P)
 2:     if is_SAT(I ∧ ¬P) then                          // unsafe check of initial state
 3:         return unsafe
 4:     O_0 := I, U := {¬P}, k := 0
 5:     while true do
 6:         while is_SAT(U) do
 7:             s := get_model()                        // s ∈ U
 8:             if UnsafeCheck(s, k) then
 9:                 return unsafe                       // counterexample found
10:         if SafeCheck(k) then
11:             return safe                             // property proved
12:         k := k + 1, O_k := ⊤
13:
14: function UnsafeCheck(s, i)
15:     while is_SAT(O_i ∧ T, s') do
16:         if i = 0 then
17:             return true
18:         t := get_predecessor(s, i)                  // (t, s) ∈ T, see Alg. 5
19:         U := U ∪ {t}
20:         if UnsafeCheck(t, i − 1) then
21:             return true
22:     c := generalize({l|l' ∈ get_UC()}, i)           // c ⊆ s, see Alg. 3
23:     O_{i+1} := O_{i+1} ∩ ¬c
24:     return false
25:
26: function SafeCheck(k)
27:     propagation(k)                                  // see Alg. 5
28:     i := 0
29:     while i < k do
30:         if not is_SAT(O_{i+1} ∧ ¬(⋁_{0≤j≤i} O_j)) then
31:             return true
32:     return false
```

until O_k is strong enough to imply P.[4] The safe check (line 25 of Alg. 1, line 26 of Alg. 2) tries to propagate the clauses in O_i to O_{i+1} and check if a fixpoint is reached. If so then the algorithm terminates. Both algorithms make use of similar additional procedures, which will be detailed in the following section, when we introduce our novel heuristics.

3 Finding *i*-good Lemmas

In this section, we introduce the concept of *i-good lemmas*, define the heuristics to steer the search towards i-good lemmas and describe the IC3 and CAR algorithms enhanced with i-good lemmas. For the sake of convenient description, we fix the input system $Sys = \langle X, Y, I, T\rangle$ and the property P to be verified. In describing the implementation of our heuristics, we shall necessarily assume that the reader has some familiarity with the low-level details of IC3 and CAR, for which we refer to [11, 17, 23]. Specifically, we shall use pseudo-code descriptions of the main components of the algorithms (Alg. 3, 4, and 5), in which the modifications required to implement our heuristics are highlighted in blue.

[4] Note that in the unsafe check, the meaning of the SAT query **is_SAT**$(O_i \wedge T, s')$ is different between CAR and IC3 (line 15 Alg. 2) so that when it is unsatisfiable the obtained clauses have different semantics.

3.1 What are *i*-good Lemmas

The over-approximate state sequence O in IC3 (resp. CAR) is a finite sequence, in which every element O_i ($0 \leq i < |O|$), namely *frame* i, is an over-approximation of the states of the system that are reachable in up to (resp. exactly) i steps from I, and which is strong enough to imply P. Such sequence O has the form of $P \wedge C$, where C is a CNF, and each clause in C is called a *lemma*. For both algorithms, the goal is that of transforming the sequence O to construct an over-approximation of all the reachable states of the system (over an unbounded horizon) that still implies P. When this happens, such over-approximation is an inductive invariant that proves P. The key idea, common to both IC3 and to CAR, is to construct the invariant *incrementally* and by reasoning in a *localized manner*, by (i) considering increasingly-long sequences of overapproximations, and by (ii) trying to propagate forward individual lemmas from a frame O_i to its successor O_{i+1}, until a fixpoint is reached[5]. The forward propagation procedure is crucial for ensuring the convergence of the algorithm in practice: for IC3 (resp. CAR), it checks whether a lemma c at frame i represents also an overapproximation of all the states reachable in up to (resp. exactly) $i+1$ steps, and therefore can be added to frame $i+1$. It is immediate to see that the successful propagation of *all* lemmas from i to $i+1$, for some i, is a sufficient condition for the termination of both IC3 and CAR with a safe result. In fact, for IC3, this is also a necessary condition.

We now introduce the notion of *i-good lemma*.

Definition 1 (i-Good Lemma). *Let c be a lemma that was added at frame i by IC3/CAR (at some previous step in the execution of the algorithm), i.e. $O_i \models c$. We say that c is i-good if c now holds also at frame $i+1$, i.e. $O_{i+1} \models c$.*

The following theorems are then consequences of the definition.

Theorem 1. IC3 *terminates with safe at frame i ($i > 0$), if and only if every lemma at frame i is i-good.*

Theorem 2. CAR *terminates with safe at frame i ($i > 0$), if every lemma at frame i is i-good.*

Such theorems provide the theoretical foundation on which we base our main conjecture: the computation of i-good lemmas can be helpful for both IC3 and CAR to accelerate the convergence in proving properties. Intuitively, an i-good lemma shows the promise of being independent of the reachability layer, and hence holds in general.

[5] The algorithms differ in the way they check reaching the fixpoint, but this difference will be ignored unless otherwise stated.

3.2 Searching for *i*-good Lemmas

Our conjecture is that there exists, on average, a positive correlation between the ratio of i-good lemmas vs the total amount of lemmas computed by IC3/CAR during generalization and the efficiency of the algorithm.

Ensuring that only i-good lemmas are produced is as hard as solving the verification problem itself, since this is essentially equivalent to synthesizing an inductive invariant which implies P. However, there are two situations in which it is easy to *identify* i-good lemmas, for both IC3 and CAR:

1. In the *propagation* procedure, if a lemma c can be successfully pushed from frame i to frame $i+1$, then c is i-good;
2. In the *generalize* procedure, if the current lemma c at frame i is generalized to a lemma $p \subseteq c$ such that $p \in O_{i-1}$, then p is $(i-1)$-good; additionally, if we can *guide the generalization* of c so that it produces p, then p becomes $(i-1)$-good.

Therefore, we do not attempt to compute only i-good lemmas, but rather, our main idea is to use some (cheap) heuristics to increase the probability of producing i-good lemmas during the normal execution of IC3 and CAR.

We exploit the above observations to design two heuristics that try to bias the search for lemmas towards those that are more likely to be i-good, which we call respectively branching and refer-skipping.

Branching. The branching strategy [26] is an important feature of modern CDCL (Conflict-Driven Clause Learning) SAT solvers [7]. Traditional scoring schemes for branching such as VSIDS and EVSIDS have been extensively evaluated in [10]. In CDCL SAT solvers, decision variables are selected according to their priority. Whenever a conflict occurs, the priority of each variable in the clause is increased. To this end, variables that have recently been involved in conflicts are more likely to be selected as decision variables.

We adopt a similar idea in our branching heuristic for IC3/CAR to bias the unsatisfiable cores produced by the SAT solver, by ordering the assumptions in SAT queries according to their score. This is based on the fact that modern SAT solvers based on CDCL apply the assumption literals in the order given by the user, and (as a consequence of how CDCL works) the unsatisfiable core produced when the formula is unsatisfiable depends on such order, with literals occurring earlier in the assumption list being more likely to be included in the core. For example, assume the SAT query is $\mathbf{is\_SAT}(\neg 1 \wedge (2 \vee \neg 3),\ 1 \wedge \neg 2 \wedge 3)$, which is unsatisfiable, then the returned UC from the SAT solver, e.g., Minisat [18, 5], will be $\{1\}$. If the order of assumptions is changed to $3 \wedge \neg 2 \wedge 1$, then the UC will be $\{3, \neg 2\}$.

Since UCs are the source for lemmas in both IC3 and CAR, the first idea of our branching heuristic is that of sorting the assumption literals in SAT queries according to *how often they occur in recent i-good lemmas*. Concretely, this is implemented as follows:

- We introduce a mapping $S_{[v]} : v \rightarrow score_v, v \in X$ from each variable to its score (priority). Initially, all variables have the same score of 0.

- Before each SAT query in which a (negated) lemma c (or its next-state version c') is part of the assumptions, c is sorted in descending order of $S_{[var(l)]}$, where $l \in c$, to give higher priority to assumption literals with higher scores. This corresponds to the calls to the function SORT(c) in the pseudo-code description of the main components of IC3 and CAR: at the beginning of UNSAFECHECK (Alg. 1 and 2), in GET_PREDECESSOR (line 6 of Alg. 4, line 6 of Alg. 5), and in GENERALIZATION (line 25 of Alg. 4, line 23 of Alg. 5).
- Whenever IC3 or CAR discovers an i-good lemma c, all the variables in c are *rewarded* by increasing their score. A lemma c is determined to be i-good either when it is propagated forward from frame i to frame $i+1$ (function PROPAGATION of Alg. 4 and 5) or when c is the result of a generalization from $d \supseteq c$ at frame $i+1$ such that c is already in frame i (function GENERALIZE, Alg. 3). In the pseudo-code, the reward steps correspond to the calls to the function REWARD(c) at line 12 of Alg. 3, line 42 of Alg. 4, and line 37 of Alg. 5. The REWARD function first decays the scores of all the variables in $S_{[v]}$ by a small amount (we multiply by 0.99 in our implementation), and then increments the score of all the variables in c (by 1 in our implementation). In order to determine whether GENERALIZE produced an i-good lemma, we also use the function GET_PARENTNODE(c) (line 3 of Alg. 3), which returns a cube p in frame $i-1$ such that $p \subseteq c$ when c belongs to frame i. (If multiple such p exist, the one with the highest score is returned).
- When performing inductive generalization of a lemma c at frame i (Alg. 3), in which c is strengthened by trying to drop literals from it as long as the result is still a valid lemma for frame i, the literals of c are sorted in increasing order of $S_{[var(l)]}$, with $l \in c$. This corresponds to the call to the function REVERSE_SORT(c) at line 2 of Alg. 3 in the pseudo-code.

Algorithm 3 Lemma Generalization of IC3/CAR

```
1:  function GENERALIZE(c, i, rec_lvl = 1)
2:      REVERSED_SORT(c)                          // sort literals in c in increasing order of priority
3:      ¬p :=GET_PARENTNODE(¬c)                   // ¬p ∈ F_{i-1}(O_{i-1}) and p ⊆ c
4:      req := p                                  // skip literals in p
5:      for each l ∈ c and l ∉ req do
6:          cm := c \ {l}
7:          if DOWN(cm, i, rec_lvl, req) then     // CTG-based dropping, see Alg. 4 and 5
8:              c := cm
9:          else
10:             req := req ∪ {l}                  // failed to drop l
11:     if c \ p = ∅ then                         // whether c is a good lemma
12:         REWARD(c)                             // raise priority of variables in c
13:     return c
```

Skipping Literals by Reference. Lemma generalization is a crucial process in IC3/CAR that affects performance significantly. Given the original lemma c to be added into frame i ($i > 0$), the GENERALIZE procedure tries to compute a new lemma g such that $g \subseteq c$ and g is also valid to be added to frame i (O_i).

Algorithm 4 Auxiliary functions for IC3

```
1:  function GET_PREDECESSOR(s, i)                          // generalization of predecessors
2:      ASSERT(is_SAT(O_i ∧ ¬s ∧ T, s'))                    // precondition: ∃t that (t, s) ∈ T
3:      μ := get_model()
4:      in := {l ∈ μ | var(l) ∈ Y}
5:      t := {l ∈ μ | var(l) ∈ X}
6:      SORT(t)                                  // sort literals in s in descending order of priority
7:      while not is_SAT(O_i ∧ in ∧ ¬s', t) do
8:          if t =get_UC() then
9:              break
10:         t :=get_UC()
11:     return t
12:
13: function DOWN(c, i, rec_lvl, req)                       // CTG-based dropping literals
14:     cex_num := 0
15:     while true do
16:         if is_SAT(I ∧ c) then
17:             return false
18:         if not is_SAT(O_i ∧ ¬c ∧ T, c') then
19:             c :={l | l' ∈ get_UC()}
20:             return true
21:         else if rec_lvl > MAX_REC_LVL then                  // MAX_REC_LVL = 3
22:             return false
23:         else
24:             cex := GET_PREDECESSOR(c, i)        // cex as a counter-example of generalization
25:             SORT(cex)                           // sort literals in s in descending order of priority
26:             if cex_num < MAX_CEX_NUM and i > 0 and not is_SAT(O_{i-1} ∧ ¬cex ∧ T, cex')
                    and not is_SAT(I ∧ cex) then                // MAX_CEX_NUM = 3
27:                 c_cex :=GENERALIZE({l | l' ∈ get_UC()}, i − 1, rec_lvl + 1)
28:                 O_k := O_k ∩ ¬c_cex, 1 ≤ k ≤ i − 1
29:                 cex_num + +
30:             else
31:                 cex_num := 0
32:                 if (c \ cex) ∩ req ≠ ∅ then
33:                     return false
34:                 c := c ∩ cex
35:
36: function PROPAGATION(k)
37:     i := 1
38:     for i < k do
39:         for ¬c ∈ O_i do
40:             if not SAT(O_i ∧ ¬c ∧ T, c') then
41:                 O_{i+1} := O_{i+1} ∩ ¬c
42:                 REWARD(c)                               // raise priority of variables in c
```

The main idea of generalization is to try to drop literals in the original lemma one by one, to see whether the left part can still be a valid lemma.

There are several generalization algorithms with different trade-offs between efficiency (in terms of the number of SAT queries) and effectiveness (in terms of the potential reduction in the size of the generalized lemma), e.g. [11, 17, 20]. More in general, there might be multiple different ways in which a lemma c can be generalized, with results of uncomparable strength (i.e. there might be both $g_1 \subseteq c$ and $g_2 \subseteq c$ such that $g_1 \not\subseteq g_2$ and $g_2 \not\subseteq g_1$).

The main idea of the refer-skipping heuristic is to bias the generalization to increase the likelihood that the result g is a $(i-1)$-good lemma. Consider the generalization of lemma $c = \neg 1 \vee 2 \vee \neg 3$ at frame i $(i > 1)$. If there is already a lemma $g = \neg 1 \vee \neg 3$ at frame $i-1$, we say that g is a *candidate* $(i-1)$*-good lemma* for the generalization of c. In order to drive the generalization of c towards g, we

Algorithm 5 Auxiliary functions for CAR

```
1:  function GET_PREDECESSOR(s, i)                          // generalization of predecessors
2:      ASSERT(is_SAT(O_i ∧ T, s'))                         // precondition: ∃t that (t, s) ∈ T
3:      μ := get_model()
4:      in := {l ∈ μ | var(l) ∈ Y}
5:      t := {l ∈ μ | var(l) ∈ X}
6:      SORT(t)                                             // sort literals in s in descending order of priority
7:      while not is_SAT(O_i ∧ in ∧ ¬s', t) do
8:          if t =get_UC() then
9:              break
10:         t :=get_UC()
11:     return t
12:
13: function DOWN(c, i, rec_lvl)                            // CTG-based dropping literals
14:     cex_num := 0
15:     while true do
16:         if not is_SAT(O_i ∧ T, c') then
17:             c :={l | l' ∈ get_UC()}
18:             return true
19:         else if rec_lvl > MAX_REC_LVL then              // MAX_REC_LVL = 3
20:             return false
21:         else
22:             cex := GET_PREDECESSOR(c, i)
23:             SORT(cex)                                   // sort literals in s in descending order of priority
24:             if cex_num < MAX_CEX_NUM and i > 0
                    and not is_SAT(O_{i-1} ∧ T, cex) then   // MAX_CEX_NUM = 3
25:                 c_cex :=GENERALIZE({l | l' ∈ get_UC()}, i − 1, rec_lvl + 1)
26:                 O_{i-1} := O_{i-1} ∩ ¬c_cex
27:                 cex_num + +
28:             else
29:                 return false
30:
31: function PROPAGATION(k)
32:     i := 1
33:     for i < k do
34:         for ¬c ∈ O_i do
35:             if not SAT(O_i ∧ T, c') then
36:                 O_{i+1} := O_{i+1} ∩ ¬c
37:                 REWARD(c)                               // raise priority of variables in c
```

blacklist the literals of g, so that GENERALIZE will never attempt to drop them from c. As such, we call g a reference for skipping generalization. In general, there might be multiple references for a given lemma. Currently, our strategy in refer-skipping is to just pick the one first found.

The implementation of refer-skipping is based on existing generalization algorithms and only needs to add less than 10 lines in the pseudo-code (see line 4-10 of Alg. 3). As shown in the algorithm, a variable set *req* is maintained to store variables that fail to be dropped so that they are not tried to be removed again later. In order to use refer-skipping, we simply initialize *req* with the variables occurring in the candidate $(i-1)$-good lemma that is returned by the GET_PARENTNODE procedure (line 3 of Alg. 3).

Finally, note that although in our pseudo-code (and in our implementation) we use the CTG algorithm of [20], the idea discussed here can be applied also to the other variants of generalization just as easily.

4 Related Work

In the field of safety model checking, after the introduction of IC3 [11], several variants have been presented: [20] presents the counterexample-guided generalization (CTG) of a lemma by blocking states that interfere with it, which significantly improves the performance of IC3; AVY [33] introduces the ideas of IC3 into IMC (Interpolant Model Checking) [25] to induce a better model checking algorithm; its upgrade version kAVY [32] uses k-induction to guide the interpolation and IC3/PDR generalization inside; [28] proposes to combine IC3/PDR with reverse IC3/PDR; the subsequent work [29] interleaves a forward and a backward execution of IC3 and strengthens one frame sequence by leveraging the proof-obligations from the other; IC3-INN [15] enables IC3 to leverage the internal signal information of the system to induce a variant of IC3 that can perform better on certain industrial benchmarks; [30] introduces under-approximation in PDR to improve the performance of bug-finding.

The importance of discovering inductive lemmas for improving convergence is first noted in [17]. In PDR terminology, inductive lemmas are the ones belonging to frame O_∞, as they represent an over-approximation of all the reachable states.

The most relevant related work is [21], where a variant of IC3 named QUIP is proposed for implementing the pushing of the discovered lemmas to O_∞. At its essence, QUIP adds the negation of a discovered lemma c as a *may*-proof-obligation, hence trying to push c to the next frame. Counterexamples of may-proof-obligations represent an under-approximation of the reachable states and are stored to disprove the inductiveness of other lemmas. In QUIP terminology, such lemmas are classified as *bad lemmas*, as they have no chance of being part of the inductive invariant. Since the pushing is not limited to the current number of frames, inductive lemmas are discovered when all the clauses of a frame can be pushed ($O_k \setminus O_{k+1} = \emptyset$ for a level k), and then added in O_∞. In QUIP terminology, lemmas belonging to O_∞ are classified as *good lemmas*, and are always kept during the algorithm. Observe that the concept of *good* lemma in [21] is a stronger version of Definition 1, which instead is *local* to a frame i and characterizes lemmas that can be propagated one frame ahead.

Both QUIP and our heuristic try to accomplish a similar task, which is prioritizing the use of already discovered lemmas during the generalization. There are however several differences: QUIP proceeds by adding additional proof-obligations to the queue and by progressively proving the inductiveness of a lemma relative to any frame. Our approach, on the other hand, is based on a cheap heuristic strategy that *locally* guides the generalization prioritizing the locally good lemmas. Some i-good lemmas computed may not be part of the final invariant and can not be pushed later; in QUIP, such lemmas would not be considered good. In our view, pushing them is not necessarily a waste of effort, because they still strengthen the frames and their presence might be necessary to deduce the final invariant. Finally, it is worth mentioning that our heuristic is much simpler to implement and integrate into different PDR-based engines.

The idea of ordering literals when performing inductive generalization is already proposed in [11] and adopted, as a default strategy, in several implementa-

tions of IC3 [3, 17, 19], yielding modest improvements on HWMCC benchmarks, however without clear trends identified (see [17, 19]). Compared to such works, our approach has two main differences. First, these heuristics favor literals occurring more frequently in all previous frames, whereas our approach is driven by the role of lemmas and prefers the variables occurring only in those are i-good. Second, our use of ordering heuristics is more pervasive: unlike in previous works, where variable ordering heuristics are only used during the lemma generalization, we use ordering everywhere the SAT results affect search direction, which makes it more effective to bias the search.

5 Evaluation

5.1 Experimental Setup

We integrated the branching and refer-skipping heuristics into three systems: the IC3Ref [3] and SimpleCAR [6] (open-source) model checkers, which implement the IC3 and (Forward and Backward[6]) CAR algorithms respectively, and the mature, state-of-the-art implementation of IC3 available inside the nuXmv model checker [12]. We make our implementations and data for reproducing the experiments available at https://github.com/youyusama/i-Good_Lemmas_MC.

Since our approach is related to QUIP [21], we include the evaluation of QUIP, and IC3 (mainly as the baseline for QUIP), as implemented[7] in IIMC [4]. We also consider the PDR implementation in the ABC model checker [1], which is state-of-the-art in hardware model checking.

Table 3. Tools and algorithms evaluated in the experiments.

Tools	Algorithms	Available Flags
IC3Ref [3]	IC3 (ic3)	-br | -rs | -sh
SimpleCAR [6]	Forward CAR (fcar)	-br | -rs | -sh
nuXmv [12]	IC3 (nuXmv)	-br | -rs | -sh
IIMC [4]	QUIP (iimc-quip)	-
IIMC [4]	IC3 (iimc-ic3)	-
ABC [1]	PDR (abc-pdr)	-

Table 3 summarizes the tested tools, algorithms, and their flags. We use the flag "-br" to enable the branching heuristic and "-rs" to enable refer-skipping. Furthermore, we evaluate also another configuration (denoted as "-sh"), in which the calls to SORT() functions in Algorithms 4 and 5 are replaced by random shuffles, thus simulating a strategy that orders variables randomly. When no flag

[6] Although there is an implementation of Backward CAR in SimpleCAR, this methodology corresponds to reverse IC3. As a result, we did not include Backward CAR in this paper and left the evaluation in future work.

[7] As far as we know, this is the only publicly available QUIP implementation.

is active, IC3Ref runs the instances with its own strategy of sorting variables, present in the original implementation.

We evaluate all the tools on 749 benchmarks, in *aiger* format, of the SINGLE safety property track of the 2015 and 2017 editions of HWMCC [8][8]. We ran the experiments on a cluster, which consists of 2304 2.5GHz CPUs in 240 nodes running RedHat 4.8.5 with a total of 96GB RAM. For each test, we set the memory limit to 8GB and the time limit to 5 hours. During the experiments, each model-checking run has exclusive access to a dedicated node.

To increase our confidence in the correctness of the results, we compare the results of the solvers to make sure they are all consistent (modulo timeouts). For the cases with unsafe results, we also check the provided counterexample with the *aigsim* tool from the Aiger package [2]. We have no discrepancies in the results, and all unsafe cases successfully pass the *aigsim* check.

5.2 Experimental Results

Overview The results of the experimental evaluation are discussed below. We first consider the aggregated results, as reported in Table 4. For each tool, we group the results obtained with the various configurations; we report the total number of benchmarks solved, distinguishing between safe and unsafe benchmarks; we also report the benchmarks gained and lost by the configurations with branching and/or refer-skipping active, relative to the baseline where branching and refer-skipping are not active. We can draw the following conclusions.

- The proposed heuristics are in general effective in improving performance. Each of the model checkers, with at least one of branching and refer-skipping active, consistently outperforms the respective baseline in terms of the number of benchmarks solved.
- The same holds within the safe instances, with the exception of refer-skipping in nuXmv that solves two safe benchmarks less than the baseline.
- The heuristics also yield a uniform improvement over the baseline in the unsafe instances.
- The combination of branching and refer-skipping gives further improvements over a single technique, with the exception of nuXmv with branching, which cumulatively solves 5 more benchmarks than nuXmv with branching and refer-skipping.
- The gain is not uniform across the instances. For example, nuXmv with branching gains 52 benchmarks (44 safe and 8 unsafe) that are not solved by nuXmv baseline, while losing 13 (safe) benchmarks. This level of variability can be expected, given a heuristic approach, but further investigation is needed to assess the underlying phenomena.

[8] From HWMCC 2019, the official format used in the competition is switched from Aiger to Btor2 [27], a format for word-level model checking. As a result, we did not include those instances in our experiments.

– The performance of using the heuristics guided by random variable ordering does not differ significantly from the baseline in terms of aggregate results. There are some differences (as expected) at the level of individual instances, especially for CAR, but no clear trend emerges overall.
– The comparison also shows that the considered systems compare well against the state-of-the-art system ABC, and QUIP; QUIP turns out to be quite inefficient and is disregarded in the following. Note that the original implementation of QUIP is not available; the fact that the available version of QUIP implemented on top of IIMC does not seem to achieve the same improvements reported in the original paper [21] (the code for which is unfortunately not available) suggests that the QUIP is far from trivial to implement. As the reference, QUIP performs even worse than the IC3 implementation in IIMC, whose performance is similar to the IC3Ref baseline, see Table 4.

Table 4. Summary of overall results among different configurations.

Configuration	#Solved	#Safe	#Unsafe	Gained(safe/unsafe)	Lost(safe/unsafe)
ic3 -br -rs	*439*	*313*	*126*	25(18/7)	6(4/2)
ic3 -br	428	302	*126*	22(15/7)	14(12/2)
ic3 -rs	430	308	122	21(17/4)	11(8/3)
ic3	420	299	121	-	-
ic3 -sh	417	297	120	9(7/2)	12(9/3)
fcar -br -rs	*444*	*319*	*125*	54(43/11)	1(0/1)
fcar -br	429	308	121	43(33/10)	5(1/4)
fcar -rs	410	295	115	23(22/1)	4(3/1)
fcar -sh	394	277	117	31(22/9)	28(21/7)
fcar	391	276	115	-	-
nuXmv -br -rs	497	353	*144*	49(39/10)	15(15/0)
nuXmv -br	*502*	*360*	142	52(44/8)	13(13/0)
nuXmv -sh	473	333	140	26(19/7)	16(15/1)
nuXmv -rs	464	327	137	7(4/3)	6(6/0)
nuXmv	463	329	134	-	-
abc-pdr	430	315	115	-	-
iimc-ic3	418	307	111	-	-
iimc-quip	377	281	96	-	-

Similar insights can be obtained from Fig. 1, which clearly shows the positive effect of improvements in performance.

Detailed statistics As shown in Table 4 and Fig. 1, nuXmv is highly optimized and has a much better performance than other open-source IC3 implementations, but enabling both heuristics is still useful to improve its overall performance by solving 34 more instances. For IC3Ref and SimpleCAR, the increased numbers of solved cases are 19 and 53, respectively. Moreover, from Table 4, nuXmv/IC3Ref/SimpleCAR is able to solve 24/14/43 more safe and 10/5/10 more unsafe instances with both heuristics.

A comparison of the performance of the tools with and without the heuristics is shown in Fig. 2. All three solvers are able to reduce their time cost when equipping with branching and refer-skipping (see the last row of the figure). Explicitly,

67.8% of the instances cost less or equal to check by 'nuXmv -br -rs', and the corresponding portions for 'ic3 -br -rs' and 'fcar -br -rs' are 77.9% and 87.0%. The variability occurs when considering only a single heuristic, which needs to be explored in the future. For example, 'fcar -br' and 'nuXmv -rs' generally cost slightly more time than 'fcar' and 'nuXmv', respectively.

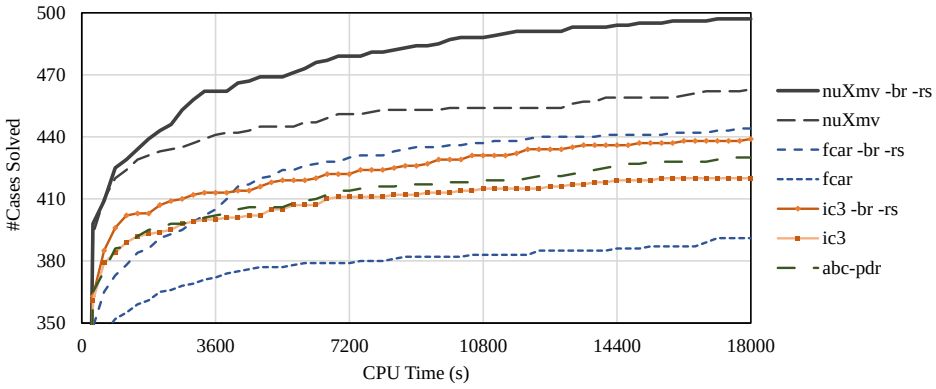

Fig. 1. Comparisons among the implementations of IC3, PDR and CAR under different configurations. (To make the figure more readable, we skip the results with a single heuristic, which are still shown in Table 4.)

According to Table 4, either branching or refer-skipping is effective for improving nuXmv, IC3Ref, and SimpleCAR. For nuXmv and SimpleCAR, branching is more useful, considering that 'nuXmv -br' (resp. 'fcar -br') solves 39 (resp. 38) more instances than 'nuXmv' (resp. 'fcar'), with 31 (resp. 32) safe and 8 (resp. 6) unsafe. For IC3Ref, the improvement with either heuristic seems relatively modest, i.e., 'ic3 -br' solves 8 more instances than 'ic3', with 3 safe and 5 unsafe, while 'ic3 -rs' solves 10 more instances than 'ic3', with 9 safe and 1 unsafe.

As listed above, 'ic3 -br -rs' loses only 6 instances that are solved by 'ic3', while 'fcar -br -rs' even loses only 1 instance that is solved by 'fcar', which indicates the performance domination of 'fcar -br -rs' over 'fcar'. For 'nuXmv -br -rs', the number of lost cases is 15, which is still modest when compared to the gain of 49. So enabling branching and refer-skipping together makes the checkers pay a limited cost. The same applies to the situations when equipping with only one single heuristic for the checkers, see Table 4.

5.3 Why do branching and refer-skipping work?

To measure why branching and refer-skipping work, we introduce sr, i.e. the **s**uccess **r**ate in computing i-good lemmas. Formally, $sr = N_g/N$ where N_g is the number of generalizations that successfully return i-good lemmas, while N is the total number of generalization calls. We instrumented the two open-source

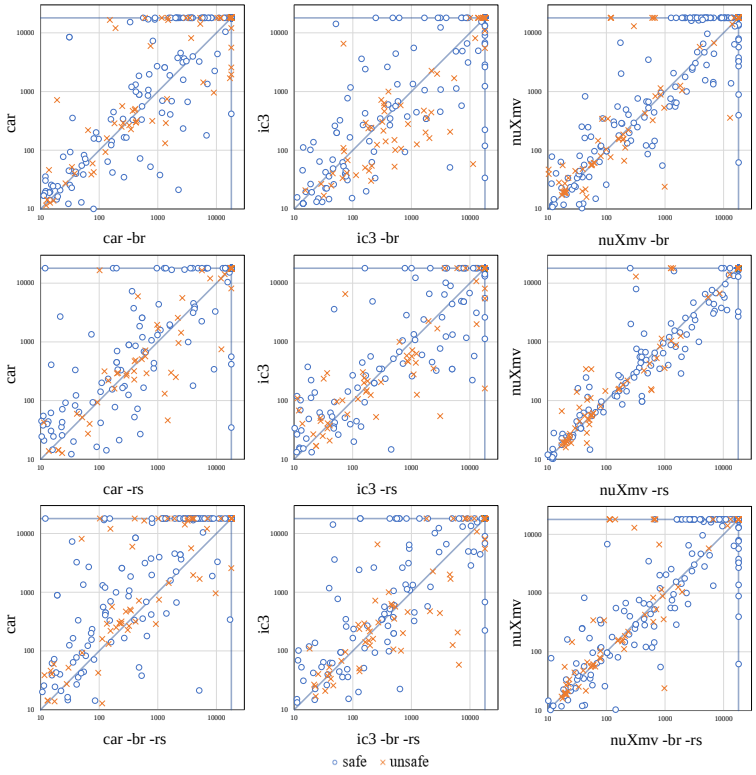

Fig. 2. Time comparison between IC3/CAR with and without two heuristics on safe-unsafe cases. The baseline is always on the y-axis. Points above the diagonal indicate better performance with the heuristics active. Points on the borders indicate timeouts (18000s).

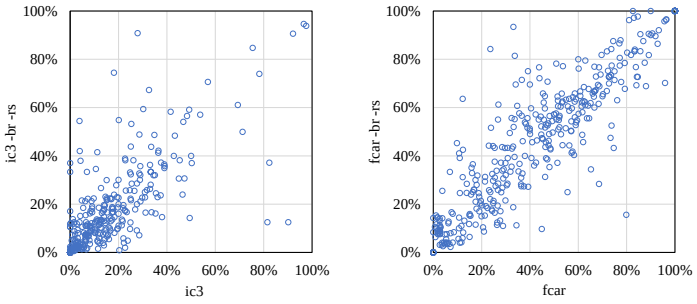

Fig. 3. Comparison on the success rate (sr) to compute i-good lemmas between IC3/CAR with and without branching and refer-skipping.

checkers IC3Ref and SimpleCAR in order to compute sr for each terminating run (including each run with/without a returned result at timeout).

- Consider the results presented in Fig. 3. The figure shows the comparison of the success rate in computing i-good lemmas between IC3/CAR with and without the heuristics. 'ic3 -br -rs' computes more i-good lemmas than 'ic3' on 54% tested instances, while 'fcar -br -rs' computes more i-good lemmas than 'fcar' on 67% tested instances, the portion of which is even higher. This supports the conjecture that enabling branching and refer-skipping makes IC3/CAR compute more i-good lemmas.
- Now consider Fig. 4. The figure shows the comparison between the deviation of success rate to compute i-good lemmas (Y axis) and the deviation of checking (CPU) time (X axis) for IC3/CAR with and without the heuristics. The meaning of each point in the plot is explained in the title of the figure. In general, the more points located in the first quadrant, the better our claim can be supported.

  Clearly, the plot for both IC3 and CAR in Fig. 4 supports the conjecture that searching more i-good lemmas can help achieve better model-checking performance (time cost).

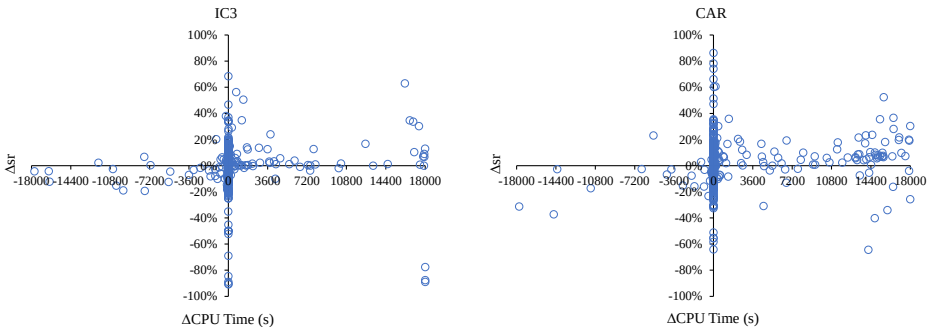

Fig. 4. Comparison between the deviation of the success rate (sr) to compute i-good lemmas (Y axis) and the deviation of checking (CPU) time (X axis) for IC3/CAR with and without the heuristics. For each instance, let the checking time of 'ic3'/'fcar' be t and that of 'ic3 -br -rs'/'fcar -br -rs' be t'. Each point has $t - t'$ as the x value and $sr' - sr$ as the y value.

Finally, we argue that computing as many i-good lemmas as possible is the direction to take to improve the performance of IC3 and its variants. branching and refer-skipping are two heuristics that can enable IC3/CAR to compute more i-good lemmas. However, there can be more efficient ways to compute i-good lemmas, which is left for our future work.

6 Conclusions and Future Work

In this paper, we proposed a heuristic-based approach to improve the performance of IC3-based safety model checking. The idea is to steer the search of the over-approximation sequence towards i-good lemmas, i.e. lemmas that can be pushed from frame i to frame $i+1$. On the one side, we attempt to control the way the SAT solver extracts the unsat cores, by privileging variables occurring in i-good lemmas (branching); on the other, we control lemma generalization by avoiding dropping literals that occur in a subsuming lemma in the previous layer (refer-skipping). The approach is very simple to implement and has been integrated into two open-source model checkers and an industrial-strength, closed-source model checker. The experimental evaluation, carried out on a wide set of benchmarks, shows that the approach yields computational benefits on all the implementations. Further analysis shows a correlation between i-good lemmas and performance improvements and suggests that the proposed heuristics are effective in finding more i-good lemmas.

In the future, we plan to investigate the reasons for performance improvement/degradation at the level of the single benchmarks. We will also attempt to integrate the proposed ideas with the ideas in QUIP, explore different kinds of heuristics, and lift this approach to the safety checking of infinite-state systems [13, 14].

Acknowledgment. We thank anonymous reviewers for their helpful comments. This work is supported by National Natural Science Foundation of China (Grant #U21B2015 and #62002118) and Shanghai Collaborative Innovation Center of Trusted Industry Internet Software. This work has been partly supported by the project "AI@TN" funded by the Autonomous Province of Trento and by the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU.

References

1. ABC. https://github.com/berkeley-abc/abc
2. AIGER Tools. http://fmv.jku.at/aiger/aiger-1.9.9.tar.gz
3. IC3Ref. https://github.com/arbrad/IC3ref
4. IIMC-QUIP. https://github.com/ryanberryhill/iimc
5. Minisat 2.2.0. https://github.com/niklasso/minisat
6. SimpleCAR. https://github.com/lijwen2748/simplecar/releases/tag/v0.1
7. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M.: Proceedings of sat competition 2022: Solver and benchmark descriptions. Department of Computer Science Series of Publications B, vol. B-2022-1 http://hdl.handle.net/10138/347211
8. Biere, A.: AIGER Format. http://fmv.jku.at/aiger/FORMAT
9. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)

10. Biere, A., Fröhlich, A.: Evaluating cdcl variable scoring schemes. In: Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18. pp. 405–422. Springer (2015)
11. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
12. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014)
13. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Ic3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 46–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
14. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools. CoRR **abs/2109.12821** (2021)
15. Dureja, R., Gurfinkel, A., Ivrii, A., Vizel, Y.: Ic3 with internal signals. In: 2021 Formal Methods in Computer Aided Design (FMCAD). pp. 63–71 (2021)
16. Dureja, R., Li, J., Pu, G., Vardi, M.Y., Rozier, K.Y.: Intersection and rotation of assumption literals boosts bug-finding. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 180–192. Springer International Publishing, Cham (2020)
17. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. pp. 125–134. FMCAD '11, FMCAD Inc, Austin, Texas (2011)
18. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing. pp. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
19. Griggio, A., Roveri, M.: Comparing different variants of the ic3 algorithm for hardware model checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **35**(6), 1026–1039 (2015)
20. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in ic3. In: 2013 Formal Methods in Computer-Aided Design. pp. 157–164. IEEE (2013)
21. Ivrii, A., Gurfinkel, A.: Pushing to the top. In: Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design. pp. 65–72. FMCAD '15, FMCAD Inc, Austin, Texas (2015)
22. Li, J., Dureja, R., Pu, G., Rozier, K.Y., Vardi, M.Y.: Simplecar: An efficient bug-finding tool based on approximate reachability. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 37–44. Springer International Publishing, Cham (2018)
23. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: Proceedings of the 36th International Conference on Computer-Aided Design. pp. 95–100. ICCAD '17, IEEE Press (2017)
24. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. Handbook of satisfiability **185** (2009)
25. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification, pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

26. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535 (2001)
27. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , btormc and boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 587–595. Springer International Publishing, Cham (2018)
28. Seufert, T., Scholl, C.: Combining pdr and reverse pdr for hardware model checking. In: 2018 Design, Automation and Test in Europe Conference and Exhibition (DATE). pp. 49–54 (2018)
29. Seufert, T., Scholl, C.: fbpdr: In-depth combination of forward and backward analysis in property directed reachability. In: Teich, J., Fummi, F. (eds.) Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019. pp. 456–461. IEEE (2019)
30. Seufert, T., Scholl, C., Chandrasekharan, A., Reimer, S., Welp, T.: Making progress in property directed reachability. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 355–377. Springer International Publishing, Cham (2022)
31. Sheeran, M., Singh, S., Stalmarck, G.: Check safety properties using induction and a SAT-solver. In: Proc. 3rd Int. Conf. on Formal Methods in Computer-Aided Design. Lecture Notes in Computer Science, vol. 1954, pp. 108–125. Springer (2000)
32. Vediramana Krishnan, H.G., Vizel, Y., Ganesh, V., Gurfinkel, A.: Interpolating strong induction. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 367–385. Springer International Publishing, Cham (2019)
33. Vizel, Y., Gurfinkel, A.: Interpolating property directed reachability. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 260–276. Springer International Publishing, Cham (2014)