Incremental, Inductive CTL Model Checking*

Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi

ECEE Department, University of Colorado at Boulder
{zyad.hassan,bradleya,fabio}@colorado.edu

Abstract. A SAT-based incremental, inductive algorithm for model checking CTL properties is proposed. As in classic CTL model checking, the parse graph of the property shapes the analysis. However, in the proposed algorithm, called IICTL, the analysis is directed by task states that are pushed down the parse tree. To each node is associated over- and under-approximations to the set of states satisfying that node's property; these approximations are refined until a proof that the property does or does not hold is obtained. Each CTL operator corresponds naturally to an incremental sub-query: given a task state, an EX node executes a SAT query; an EU node applies IC3; and an EG node applies FAIR. In each case, the query result provides more general information than necessary to satisfy the task. When a query is satisfiable, the returned trace is generalized using forall-exists reasoning, during which IC3 is applied to obtain new reachability information that enables greater generalization. When a query is unsatisfiable, the proof provides the generalization. In this way, property-directed abstraction is achieved.

1 Introduction

Incremental, inductive verification (IIV) algorithms construct proofs by generating lemmas based on concrete hypothesis states. Through inductive generalization, a lemma typically provides significantly more information than is required to address the hypotheses. A principle of IIV is that each lemma holds *relative to* previously generated lemmas, hence the term *incremental*, so that the difficulty of lemma generation is fairly uniform throughout execution. In this way, property-directed abstraction is achieved. The safety model checker IC3 [3, 4] and the model checker FAIR [6] for analyzing ω-regular properties are both incremental, inductive model checkers. IC3 generates stepwise relatively inductive clauses in response to states that lead to property violations. FAIR generates inductive information about reachability and SCC-closed sets in response to sets of states that together satisfy every fairness constraint. This paper describes an incremental, inductive model checker, IICTL, for analyzing CTL properties of finite state systems, possibly with fairness constraints.

An investigation into an IIV model checker for CTL properties is important for several reasons. First, CTL is a historically significant specification language. Second, some properties like *resetability* ($\mathsf{AG}\,\mathsf{EF}\,p$ in CTL) require branching time

* Work supported in part by the Semiconductor Research Corporation under contracts GRC 1859 and GRC 2220.

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 532–547, 2012.

semantics. Third, on properties in the fragment common to CTL and LTL, traditional CTL algorithms are sometimes superior to traditional LTL algorithms. CTL model checking is inherently hierarchical in that a CTL property can be analyzed according to its parse DAG. In the context of IIV, the strategy that IICTL applies to such properties is different than that applied by FAIR. Finally, CTL offers a conceptual challenge that previous IIV algorithms, IC3 and FAIR, do not address: branching time semantics. In particular, CTL motivates generalizing counterexample traces in addition to using proof-based generalization.

IICTL builds on traditional parse DAG-based analyses, except that it eschews the standard global, or bottom-up, approach in favor of a task-directed strategy. Beginning with the initial states for the root node, task states—which, as in previous IIV algorithms, are concrete states of the system—are pushed down the DAG, directing a node to *decide* whether those states satisfy its associated subformula. In the process of making a decision, a node can in turn generate a set of tasks for its children, and so on. Depending on the root operator of the node, it applies a SAT solver (EX), a safety model checker such as IC3 (EU), or a fair cycle finder such as FAIR (EG), to investigate the status of the task states. Once it reaches a conclusion, it generalizes the witness—either a proof or a counterexample trace—to provide as much new information as possible.

The first approaches to SAT-based CTL model checking [1, 13] were global algorithms that leveraged the ability of CNF formulae and Boolean circuits to be reasonably sized in some cases when BDDs are not. They differ from IICTL, which is an incremental, local algorithm. McMillan [13] first proposed an efficient technique for quantifier elimination that is related to the algorithm of Section 3.2, but is not driven by a trace to be generalized. The idea of creating an unsatisfiable query to generalize a solution to a satisfiable one (used in (15) and (20) in Section 3.2) comes from [16] and is present also in [7]. A few attempts [20, 19, 14] have been made to extend bounded model checking to branching time. They are all restricted to universal properties, though, and they have not received an extensive experimental evaluation. Their effectiveness thus remains unclear.

After preliminaries in Section 2, Section 3 describes IICTL in detail. Section 4 presents the results of a prototype implementation of IICTL within the IImc model checker [11].

2 Preliminaries

A *finite-state system* is represented as a tuple $S : (\overline{i},\ \overline{x},\ I(\overline{x}),\ T(\overline{x}, \overline{i}, , \overline{x}'),\ \mathcal{B})$ consisting of primary inputs $\overline{i}$, state variables $\overline{x}$, a propositional formula $I(\overline{x})$ describing the initial configurations of the system, a propositional formula $T(\overline{x}, \overline{i}, \overline{x}')$ describing the transition relation, and a set $\mathcal{B} = \{B_1(\overline{x}), \ldots, B_\ell(\overline{x})\}$ of Büchi fairness constraints.

Primed state variables $\overline{x}'$ represent the next state. A state of the system is an assignment of Boolean values to all variables $\overline{x}$ and is described by a *cube* over $\overline{x}$, which, generally, is a conjunction of literals, each *literal* a variable or its negation. An assignment s to all variables of a formula F either satisfies the formula, $s \models F$, or falsifies it, $s \not\models F$. If s is interpreted as a state and $s \models F$,

then s is an *F-state*. A formula F *implies* another formula G, written $F \Rightarrow G$, if every satisfying assignment of F satisfies G.

The transition structure is assumed to be complete. That is, every state has at least one successor on every input: $\forall \overline{x}, \overline{i} \,.\, \exists \overline{x}' \,.(\overline{x}, \overline{i}, \overline{x}') \models T$. A *path* in S, $s_0, s_1, s_2, \ldots$, which may be finite or infinite in length, is a sequence of states such that for each adjacent pair (s_i, s_{i+1}) in the sequence, $\exists \overline{i}.(s_i, \overline{i}, s'_{i+1}) \models T$. If $s_0 \models I$, then the path is a *run* of S. A state that appears in some run of the system is *reachable*. A path $s_0, s_1, s_2, \ldots$ is *fair* if, for every $B \in \mathcal{B}$, infinitely many s_i satisfy B, $s_i \models B$; if $s_0 \models I$ then it is a *fair run* or *computation* of S.

Computational Tree Logic (CTL [8, 15]) is a branching-time temporal logic. Its formulae are inductively defined over a set A of atomic propositions. Every atomic proposition is a formula. In addition, if φ and ψ are CTL formulae, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\mathsf{EX}\,\varphi$, $\mathsf{E}\,\psi\,\mathsf{U}\,\varphi$, and $\mathsf{EG}\,\varphi$. Additional operators are defined as abbreviations. In particular, $\mathsf{EF}\,\varphi$ abbreviates $\mathsf{E}(\varphi \vee \neg\varphi)\,\mathsf{U}\,\varphi$, $\mathsf{AX}\,\varphi$ abbreviates $\neg\,\mathsf{EX}\,\neg\varphi$, $\mathsf{AG}\,\varphi$ abbreviates $\neg\,\mathsf{EF}\,\neg\varphi$, and $\mathsf{AF}\,\varphi$ abbreviates $\neg\,\mathsf{EG}\,\neg\varphi$. A model of a CTL formula is a pair $M = (S, \mathcal{V})$ of a finite-state system S and a valuation $\mathcal{V}$ of the atomic propositions as subsets of states of S. Satisfaction of a CTL formula at state s_0 of M is then defined as follows:

$M, s_0 \models a$	iff $s_0 \in \mathcal{V}(a)$ for $a \in A$
$M, s_0 \models \neg\varphi$	iff $M, s_0 \not\models \varphi$
$M, s_0 \models \varphi \wedge \psi$	iff $M, s_0 \models \varphi$ and $M, s_0 \models \psi$
$M, s_0 \models \mathsf{EX}\,\varphi$	iff $\exists$ a fair path $s_0, s_1, \ldots$ of S such that $M, s_1 \models \varphi$
$M, s_0 \models \mathsf{EG}\,\varphi$	iff $\exists$ a fair path $s_0, s_1, \ldots$ of S such that for $i \geq 0$, $M, s_i \models \varphi$
$M, s_0 \models \mathsf{E}\,\psi\,\mathsf{U}\,\varphi$	iff $\exists$ a fair path $s_0, s_1, \ldots$ of S such that there exists $i \geq 0$ for which $M, s_i \models \varphi$, and for $0 \leq j < i$, $M, s_j \models \psi$.

Then $M \models \varphi$ if $\forall s \,.\, s \models I \Rightarrow M, s \models \varphi$. That is, M models formula φ if all its initial states do. In model M, the set of states that satisfy φ is written $[\![\varphi]\!]$.

That every CTL formula is interpreted as a set of states makes model checking easier than for the more expressive CTL*. Working bottom-up on the parse graph of φ, the standard symbolic CTL model checking algorithm [12] annotates each node with a set of states. Boolean connectives are dealt with in the obvious way, while temporal operators are handled with fixpoint computations. The bottom-up approach is also known as *global* model checking. In contrast, *local* model checking [10, 17, 2, 9] proceeds top-down. A local model checker starts from the goal of proving that initial state s satisfies φ and applies inference rules to reformulate the goal as a list of subgoals in terms of subformulae of φ and states in the vicinity of s. While local model checking can sometimes prove a property without examining most of a system's states, in its basic formulation it does not play to the strengths of symbolic algorithms. For that reason, local model checkers for finite-state systems tend to employ explicit search.[1]

[1] Some BDD-based model checkers incorporate elements of local algorithms. For instance, the CTL model checker in VIS [18] uses top-down *early termination conditions* to define conditions that a safe approximation of a set of states must satisfy. However, it is still fundamentally a bottom-up algorithm.

Table 1. Initial bounds for IICTL

ψ_v	L_v	U_v
$a \in A$	$\mathcal{V}(a)$	$\mathcal{V}(a)$
$\neg\psi_i$	$\neg U_i$	$\neg L_i$
$\psi_i \wedge \psi_j$	$L_i \wedge L_j$	$U_i \wedge U_j$

ψ_v	L_v	U_v
$\mathsf{EX}\,\psi_i$	$\bot$	$\top$
$\mathsf{E}\,\psi_j\,\mathsf{U}\,\psi_i$	L_i	$U_i \vee U_j$
$\mathsf{EG}\,\psi_i$	$\bot$	U_i

3 Algorithm

The input to IICTL consists of a model $M = (S, \mathcal{V})$ and the parse graph of a CTL formula φ. Each node of the parse graph is a natural number v and is labeled with a token from φ. Node 0 is the root of the DAG. The formula rooted at v is denoted by ψ_v, so that, in particular, $\psi_0 = \varphi$. IICTL annotates each node v with two propositional formulae over the state variables: U_v and L_v, with which an upper bound formula $\mathcal{U}_v$ and a lower bound formula $\mathcal{L}_v$, discussed later, approximate the satisfying set $[\![\psi_v]\!]$ of the formula ψ_v in M. Initial approximations are computed as shown in Table 1. A global approximation of the states of S reachable from the initial states is maintained as inductive propositional formula R. Initially, $R = \top$; that is, all states are presumed reachable.

Throughout execution, IICTL maintains the following invariant:

$$[\![R \wedge U_v \wedge L_v]\!] \subseteq [\![R \wedge \psi_v]\!] \subseteq [\![R \wedge U_v]\!] \,. \tag{1}$$

All states of the left set definitely satisfy ψ_v; all states not in the right set definitely do not satisfy ψ_v or are unreachable. A state s of the system S such that $s \models R \wedge U_v$ but $s \not\models R \wedge U_v \wedge L_v$—together, $s \models R \wedge U_v \wedge \neg L_v$—is *undecided* for ψ_v. The algorithm incrementally refines the approximations by considering undecided states until either no initial state of S is undecided for φ, proving $M \models \varphi$, or an initial state $\hat{s}$ is found such that $\hat{s} \not\models U_0$, proving $M \not\models \varphi$.

Let $\mathcal{L}_v = R \wedge U_v \wedge L_v$ designate the *lower bound* states: those states that are known to satisfy ψ_v. Let $\mathcal{U}_v = R \wedge U_v$ designate the *upper bounds* states: those states that are not known not to satisfy ψ_v. Invariant (1) is then written $[\![\mathcal{L}_v]\!] \subseteq [\![R \wedge \psi_v]\!] \subseteq [\![\mathcal{U}_v]\!]$. Finally, let $\mathcal{A}_v = \mathcal{U}_v \wedge \neg\mathcal{L}_v = R \wedge U_v \wedge \neg L_v$ designate the undecided states of node v.

Section 3.1 describes the essential structure of IICTL in detail. Section 3.2 introduces *forall-exists generalization*, which is applied to counterexample traces. Then Section 3.3 describes two important refinements to the basic algorithm, while Section 3.4 describes the additions for handling fairness constraints.

3.1 An Outline of IICTL

If ever $I \wedge \neg U_0$ becomes satisfiable, then IICTL concludes that $M \not\models \varphi$: not even the over-approximation $\mathcal{U}_0$ of φ contains all I-states, so neither can φ itself. If instead $I \wedge \neg(L_0 \wedge U_0)$ becomes unsatisfiable, then $M \models \varphi$: the under-approximation $\mathcal{L}_0$ of φ contains all I-states, so φ itself must as well.

Otherwise, one or more initial undecided states must be *decided*. At the top level, a witness s to the satisfiability of $I \wedge U_0 \wedge \neg L_0$ is undecided; it is decided by calling the recursive function decide with arguments s and 0, the root of the parse tree of φ, which eventually returns true if $M, s \models \varphi$ and false otherwise. In general, decide(t, v) return true iff $M, t \models \psi_v$. A call to decide(t, v) can update L_v or U_v (or both) so that state t becomes decided for ψ_v; moreover, the call can trigger a cascade of recursive calls that update the bounds of descendants of v and, crucially, may decide many states besides t. Within the tree, the over-approximating reachability set R becomes relevant: a state t is undecided at node v if it satisfies $\mathcal{A}_v$. The pseudocode for decide listed in Figure 1 provides structure to the following discussion.

Boolean Nodes. According to Table 1, no state can be undecided for a propositional node because the initial approximations are exact; therefore, in the case that v is a propositional node, one of the conditions of lines 2–3 holds.

If $\psi_v = \psi_u \wedge \psi_w$, the following invariant is maintained:

$$U_v = U_u \wedge U_w \quad \text{and} \quad L_v = L_u \wedge L_w \ . \tag{2}$$

If t is undecided at entry, then recurring on nodes u and w decides t for v (line 6). The **update** statement (line 5; also lines 7, 20, and 32) indicates that L_v and U_v should be updated whenever a child's bound is updated during recursion. It does not express an invariant.

If $\psi_v = \neg\psi_u$, the following invariant is maintained:

$$U_v = \neg(L_u \wedge U_u) \quad \text{and} \quad L_v = \neg U_u \ . \tag{3}$$

If t is undecided at entry, then recurring on node u decides t for v (line 8).

EX Nodes. If $\psi_v = \mathsf{EX}\,\psi_u$, then the undecided question is whether t has a successor satisfying ψ_u. IICTL executes two SAT queries in order to answer this question. First, it executes an *upper bound query*. Naively, this query is $t \wedge T \wedge U'_u$, which asks whether t has a U_u-successor. However, for better generalization, the following is used instead (line 10):

$$t \wedge \mathcal{U}_v \wedge T \wedge U'_u \ . \tag{4}$$

If unsatisfiable, the core reveals cube $\bar{t} \subseteq t$ such that all $\bar{t}$-states (including t) lack U_u-successors (and thus ψ_u-successors) or are unreachable. U_v is then updated to $U_v \wedge \neg\bar{t}$ (line 11)—no $\bar{t}$-state is a ψ_u-state (or it is an unreachable ψ_u-state).

However, if query (4) is satisfiable, the witness reveals successor U_u-state s (line 13). A *lower bound query* is executed next (line 14):

$$t \wedge T \wedge L'_u \wedge U'_u \ . \tag{5}$$

If satisfiable, then t itself has been decided: it definitely has a ψ_u-successor, since it has a $(U_u \wedge L_u)$-successor (recall invariant (1)). Forall-exists generalization

```
bool decide(t : state, v : node):                                              1
  if t ⊨ R ∧ U_v ∧ L_v: return true {already decided: M,t ⊨ ψ_v}                2
  if t ⊭ R ∧ U_v: return false {already decided: M,t ⊭ ψ_v}                     3
  match ψ_v with:                                                              4
  ψ_u ∧ ψ_w: [update L_v, U_v := L_u ∧ L_w, U_u ∧ U_w]                          5
    return decide(t, u) ∧ decide(t, w)                                         6
  ¬ψ_u: [update L_v, U_v := ¬U_u, ¬(L_u ∧ U_u)]                                7
    return ¬decide(t, u)                                                       8
  EX ψ_u:                                                                      9
    if t ∧ U_v ∧ T ∧ U'_u is unsat: {with t̄ ⊆ t from core}                      10
      U_v := U_v ∧ ¬t̄                                                          11
      return false                                                             12
    else: {with t–successor s}                                                 13
      if t ∧ T ∧ L'_u ∧ U'_u is sat:                                            14
        L_v := L_v ∨ generalize(t)                                             15
        return true                                                            16
      else:                                                                    17
        decide(s, u)                                                           18
        return decide(t, v)                                                    19
  E ψ_u U ψ_w: [update L_v, U_v := L_v ∨ L_w, U_v ∧ (U_u ∨ U_w)]                20
    if ¬(t ∧ U_w is sat or reach(S, U_u ∧ U_v ∧ R ∧ U'_v, t, U_w)):            21
      U_v := U_v ∧ ¬P                                  {with proof P}          22
      return false                                                             23
    else: {with trace s_0 = t, s_1, ..., s_n}                                  24
      if t ∧ L_w is sat or reach(S, L_u ∧ U_v ∧ U'_v, t, L_v ∧ U_v):            25
        L_v := L_v ∨ generalize(r̄)                     {with trace r̄}          26
        return true                                                            27
      elif decide(s_i, u), 0 ≤ i < n, and decide(s_n, w) are true:             28
        L_v := L_v ∨ generalize(s_0,...,s_n)                                   29
        return true                                                            30
      else: return decide(t, v)                                                31
  EG ψ_u: [update U_v := U_v ∧ U_u]                                            32
    if ¬fair(S, U_v ∧ R ∧ U'_v, t): {with assertion P}                          33
      U_v := U_v ∧ ¬P                                                          34
      return false                                                             35
    else: {with trace s_0 = t, ..., s_k, ..., s_n, s_k}                        36
      if fair(S, L_u ∧ U_v ∧ R ∧ L'_u ∧ U'_v, t): {with trace r̄}               37
        L_v := L_v ∨ generalize(r̄)                                             38
        return true                                                            39
      elif decide(s_i, u), 0 ≤ i ≤ n, are true:                                40
        L_v := L_v ∨ generalize(s_0,...,s_n)                                   41
        return true                                                            42
      else: return decide(t, v)                                                43
```

Fig. 1. Basic version of the main recursive function

(Section 3.2) then produces a cube $\bar{t} \subseteq t$ of states that definitely have ψ_u-successors (or are unreachable), and L_v is updated to $L_v \vee \bar{t}$ (line 15). If the query is unsatisfiable (line 17), then apparently state s is undecided for u. In this case, decide(s, u) is called (line 18), which results in updates to at least one

of U_u and L_u, which is a form of progress. The entire process iterates until t is decided (line 19).

EU Nodes. An EU-node maintains the following invariant:

$$[\![\mathcal{L}_w]\!] \subseteq [\![\mathcal{L}_v]\!] \ , \ [\![\mathcal{L}_v]\!] \subseteq [\![\mathcal{L}_u]\!] \cup [\![\mathcal{L}_w]\!] \ , \ [\![\mathcal{U}_w]\!] \subseteq [\![\mathcal{U}_v]\!] \ , \ [\![\mathcal{U}_v]\!] \subseteq [\![\mathcal{U}_u]\!] \cup [\![\mathcal{U}_w]\!] \ . \quad (6)$$

If $\psi_v = \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$, then the undecided question is whether t has a ψ_u-path to a ψ_w-state. To answer this question, it executes two reachability queries using an engine capable of returning counterexample traces and inductive proofs, such as IC3 [3, 4]. Let $\mathsf{reach}(S, C, F, G)$ be a function that accepts a system S, a set of constraints $C(\overline{x}, \overline{x}')$ on the transition relation, an initial condition F, and a target G; and that returns either a counterexample run from an F-state to a G-state, or an assertion $P(x)$, inductive relative to C, separating F from G.

The *upper bound query* asks whether t leads to a U_w-state (line 21). First, the query $t \wedge U_w$ determines if t is itself a U_w-state. If not, then the following query determines if it can reach a U_w-state via a U_u-path:

$$\mathsf{reach}(S,\ U_u \wedge U_v \wedge R \wedge U_v',\ t,\ U_w) \ . \quad (7)$$

The transition relation constraint $U_u \wedge U_v \wedge R \wedge U_v'$ mixes the necessary (U_u) with the optimizing ($U_v \wedge R \wedge U_v'$). If the query is unsatisfiable, the returned inductive proof P shows that no U_w-state can be reached via a potentially reachable $U_u \wedge U_v$-path, deciding at least t and informing the update of U_v to $U_v \wedge \neg P$ (line 22). If either of the query $t \wedge U_w$ or query (7) is satisfiable, let $s_0 = t, s_1, \ldots, s_n$ be the returned counterexample trace (line 24).

Lower bound queries are executed next (line 25). First, decide asks if t is itself a ψ_w-state via the query $t \wedge \mathcal{L}_w$. If not, it asks whether t can reach a known ψ_v-state via a known ψ_u-path:

$$\mathsf{reach}(S,\ \mathcal{L}_u \wedge U_v \wedge U_v',\ t,\ L_v \wedge U_v) \ . \quad (8)$$

In this version, the target set has those states that are known to have ψ_u-paths to ψ_w-states. If either case is satisfiable, t is decided for v: it has a ψ_u-path to a ψ_w-state. Forall-exists generalization (Section 3.2) produces a set of states F, including t, that definitely have ψ_u-paths to ψ_w-states or are unreachable, with which L_v is updated (line 26).

However, if the query is unsatisfiable, then attention returns to the trace $s_0, \ldots, s_n$ of the upper bound query (7) to decide whether its states satisfy the appropriate subformulae (lines 28–31). Each s_i, $0 \leq i < n$, is queried for node u, and s_n is queried for node w. If all states of the trace[2] are decided positively (line 28), then t is decided positively for v; therefore, L_v is expanded by the generalization of the trace (line 29). If one of the states is decided negatively, the upper and lower bound queries are iterated until t is decided (line 31): either a trace is found, or the nonexistence of such a trace is proved.

[2] Section 3.3 refines this state-by-state treatment to the level of traces.

EG Nodes. An EG-node maintains the following invariant:

$$[\![\mathcal{L}_v]\!] \subseteq [\![\mathcal{L}_u]\!] \quad \text{and} \quad [\![\mathcal{U}_v]\!] \subseteq [\![\mathcal{U}_u]\!] \ . \tag{9}$$

If $\psi_v = \mathsf{EG}\,\psi_u$, then the undecided question is whether there exists a reachable fair cycle all of whose states are ψ_u-states. To answer this question, it executes two *fair cycle queries* using an engine capable of returning (1) fair cycles and (2) inductive reachability information describing states that lack a reachable fair cycle. FAIR is one such engine [6]. Let $\mathsf{fair}(S, C, F)$ be a function that accepts a system S, possibly with fairness constraints $\{B_1, \ldots, B_\ell\}$, a set of constraints $C(\overline{x}, \overline{x}')$ on the transition relation, and an initial condition F; and that returns either an F-reachable fair cycle, or an inductive assertion P, where $F \Rightarrow P$, describing a set of states that lack reachable fair cycles.

The *upper bound query* asks whether a reachable fair cycle whose states satisfy U_u exists. The constraint on the transition relation uses U_v because states of a counterexample should potentially be $\mathsf{EG}\,\psi_u$ states (line 33):

$$\mathsf{fair}(S,\ U_v \wedge R \wedge U_v',\ t) \ . \tag{10}$$

If the query is unsatisfiable, the returned inductive assertion P describes states, including t, that do not have reachable fair cycles (line 33); hence, U_v is updated to $U_v \wedge \neg P$ (line 34). Otherwise, a reachable fair cycle $s_0 = t, \ldots, s_k, \ldots, s_n, s_k$ is obtained (line 36).

Before exploring the trace, a *lower bound query* is executed (line 37) to determine whether a reachable fair $\mathcal{L}_u$-cycle exists[3]:

$$\mathsf{fair}(S,\ \mathcal{L}_u \wedge \mathcal{L}_u',\ t) \ . \tag{11}$$

If it is satisfiable, the resulting run is generalized (Section 3.2) to a formula F, and L_v is updated to $L_v \vee F$ (line 38).

Otherwise, the reachable fair cycle from query (10) is considered (line 40). If all s_i are ψ_u-states, decide finishes as with a satisfiable lower bound query (lines 41–42). Otherwise, the exploration updates U_v, so that some progress is made, and the process is iterated (line 43).

Even if generalize were to return what it is given, the sound updates to L_v (lines 15, 26, 29, 38, 41) and U_v (lines 11, 22, 34), combined with the progress guaranteed by each call to decide, make the basic version of IICTL correct.

Theorem 1. *IICTL terminates and returns* **true** *iff* $M \models \varphi$.

Example 1. Consider resetability, $\varphi = \mathsf{AG}\,\mathsf{EF}\,p = \neg\,\mathsf{EF}\,\neg\,\mathsf{EF}\,p$, whose parse graph, with initial upper and lower bounds is shown in Figure 2. Because initial states are undecided for 0, IICTL chooses some initial state s and calls $\mathsf{decide}(s, 0)$, which in turn calls $\mathsf{decide}(s, 1)$. To determine if s is a ψ_1-state, decide queries a safety model checker for the existence of a path from s to U_2, i.e., to a $\neg p$-state.

[3] Note that the fair cycle query could potentially be avoided by asking if known ψ_v-states are reachable from t via a ψ_u-path: $\mathsf{reach}(S,\ \mathcal{L}_u \wedge \mathcal{L}_u',\ t,\ \mathcal{L}_v)$.

If none exist, inductive proof P is returned, and U_1 is updated by $\neg P$. If counterexample trace $s, \ldots, t$ is found, decide asks whether a path from s to L_2 exists, which is currently impossible. The disagreement between U_2 and L_2 on t triggers calls to decide(t, 2) and decide(t, 3). With equal bounds for node 4, only one reachability query is needed. If t cannot reach p (case 1), the inductive proof eliminates t from U_3 and adds it to L_2. Then s can reach a ψ_2-state, deciding s for 1 positively, and $s, \ldots, t$ are added to L_1. Finally, s is removed from node 0, indicating failure of the property.

Otherwise (case 2), the discovered trace at node 3 is generalized to F, included in L_3, and eliminated from U_2. Then the upper bound reachability query of node 1 is repeated asking for the existence of a path from s to a $\neg p \wedge \neg F$-state. The procedure continues until either case 1 occurs (failure), or until this query fails, establishing at least that $s \models \psi_0$. Then decide is invoked again for node 1 with a remaining undecided initial state if any exist, or success of the property is declared.

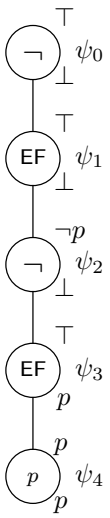

Fig. 2. AG EF p

3.2 Forall-Exists Generalization

Proofs from upper bound queries provide generalization in one direction: unsatisfiable cores for EX-nodes, inductive unreachability proofs for EU-nodes, and inductive reachability information from fair cycle queries for EG-nodes. While there are some techniques that IICTL applies to improve inductive proofs—proof strengthening, weakening, and shrinking—they have been discussed in the context of FAIR [6]. An essential aspect of making IICTL work in practice is the ability to generalize from counterexample traces. A first approach, given trace $s_0, i_0, s_1, i_1, \ldots, s_{n-1}, i_{n-1}, s_n$ with interleaved states and primary input values, is to use the unsatisfiable cores of the query $s_j \wedge i_j \wedge T \wedge \neg s'_{j+1}$ to reduce s_j to a subcube, with decreasing j [16, 7]. For greater generalization power, *forall-exists generalization* is introduced in this section. While the overall idea is similar for the three operators EX, EU, and EG, details differ.

The overall idea of forall-exists trace generalization is to (1) select a cube c of the trace, (2) flip a literal of c to obtain $\bar{c}$, and (3) decide whether all $\mathcal{A}_v \wedge \bar{c}$-states are ψ_v-states. If they are, c can be replaced with the resolvent of c and $\bar{c}$, that is, the cube obtained by dropping the literal of step (2). This process continues until no further literal of the trace can be dropped. During generalization, it is assumed that all states described by the current trace are L_v-states, so that an improvement of one cube can lead to improvements of other cubes. Hence, literals can be tried multiple times.

Selecting the cube (step (1)) and one of its literals (step (2)) can be heuristically guided. The following describes step (3) of the procedure, where $\bar{c}$ is a candidate cube obtained by flipping a literal of a cube from the current trace.

EX Nodes. Let $\psi_v = \mathsf{EX}\,\psi_u$, and let $\bar{c}$ be a cube describing a set of states. If

$$\forall \overline{x} \,.\, \bar{c}(\overline{x}) \wedge \mathcal{A}_v(\overline{x}) \rightarrow \exists \overline{i}, \overline{x}' \,.\, T(\overline{x}, \overline{i}, \overline{x}') \wedge \mathcal{L}_u(\overline{x}') \ , \tag{12}$$

then $\bar{c}$ can be added to L_v while maintaining invariant (1).

The challenge with (12) is quantifier alternation. Rather than using a general QBF solver, IICTL adopts a strategy in which two queries are executed iteratively. The first SAT query is the following:

$$\bar{c} \wedge \mathcal{A}_v \wedge T \wedge \neg \mathcal{L}'_u \ . \tag{13}$$

It asks whether all successors of $\bar{c}$-states are $\mathcal{L}_u$ states. If the query is unsatisfiable, then no undecided $\bar{c}$-state has a successor outside of $\mathcal{L}_u$. The states in $\bar{c} \wedge \neg \mathcal{A}_v$ can be added freely to L_v. Those in $\bar{c} \wedge \mathcal{A}_v$ have all their successors in $\mathcal{L}_u$; hence, they satisfy ψ_v and can be added to L_v by updating it to $L_v \vee \bar{c}$.

If, however, query (13) is satisfiable, then there exists an undecided $\bar{c}$-state s with at least one successor outside of $\mathcal{L}_u$. The second SAT query establishes whether all of s's successors are $\neg\mathcal{L}_u$-states:

$$s \wedge T \wedge \mathcal{L}'_u \ . \tag{14}$$

If the query is satisfiable, then there exists s-successor state t and input j such that $t \models \mathcal{L}_u$ and $(s, j, t') \models T$. In this case, the following query is unsatisfiable:

$$s \wedge \mathcal{A}_v \wedge j \wedge T \wedge \neg \mathcal{L}'_u \ . \tag{15}$$

The set of literals s that do not appear in the unsatisfiable core can be dropped from s to obtain cube $\bar{s}$, which describes the set of states that either are not in $\mathcal{A}_v$ or go to $\mathcal{L}_u$ under input j; these states can therefore be added to L_v, yielding $L_v \vee \bar{s}$. While query (15) is unsatisfiable even without the conjunction of $\mathcal{A}_v$, the presence of $\mathcal{A}_v$ allows generalizations of s that include $\neg\mathcal{A}_v$-states.

If query (14) is unsatisfiable, s is considered a *counterexample to generalization* (CTG). It explains why $\bar{c}$ cannot be added to L_v at this time: it is known that s lacks an $\mathcal{L}_u$-successor and thus undecided whether s has a ψ_u-successor. Hence, s remains undecided for v. However, all is not lost: the question remains whether s is even reachable. Because generalization is unnecessary for correctness but necessary for (practical) completeness, answering this question requires balancing computational costs against the potential benefits of greater generalization. There are three reasonable approaches to addressing the question: (1) ignore it, obtaining immediate speed at the cost of generalization; (2) apply a semi-decision procedure for reachability, such as the MIC procedure of FSIS and IC3 [5, 4]; (3) apply a full reachability procedure such as IC3. In the latter two cases, proofs of unreachability refine R, which strengthens all IICTL queries, in addition to making s irrelevant to the current generalization attempt.

With approach (3), even in the case that IC3 finds that s is reachable, the truly inductive clauses generated during the analysis are added to R, yielding new information. Furthermore, s is added to a set of states known to be reachable. Henceforth, whenever a cube $\bar{c}$ is considered as part of generalization at

some node v, $s \in \bar{c}$ is first tested; if so, then query (14) is immediately applied. If this query is satisfiable, then s is marked as henceforth irrelevant for generalizations at node v. This reuse of known reachable states during generalization significantly mitigates the cost of approach (3) on some benchmarks.

EU and EG Nodes. Let node v be such that either $\psi_v = \mathsf{E}\,\psi_u\,\mathsf{U}\,\psi_w$ or $\psi_v = \mathsf{EG}\,\psi_u$. In both cases, the generalization queries are motivated by the following:

$$\forall \overline{x}\,.\,\bar{c}(\overline{x}) \wedge \mathcal{A}_v(\overline{x}) \rightarrow \mathcal{L}_u(\overline{x}) \wedge \exists \overline{i}, \overline{x}'\,.\,T(\overline{x}, \overline{i}, \overline{x}') \wedge \mathcal{L}_v(\overline{x}') \ . \tag{16}$$

Any $(\bar{c} \wedge \mathcal{A}_v)$-state must be a ψ_u-state, motivating the $\mathcal{L}_u(\overline{x})$ term.[4] Additionally, it must have a ψ_v-successor, motivating the $\mathcal{L}_v(\overline{x})$ term.

To address (16) without a QBF solver, several queries are executed iteratively. First, $\neg\psi_u$-states are addressed with the following query:

$$\bar{c} \wedge \mathcal{A}_v \wedge \neg\mathcal{L}_u \ . \tag{17}$$

If satisfiable, the indicated CTG can be analyzed for reachability. A reachable CTG ends consideration of $\bar{c}$. Once query (17) becomes unsatisfiable, focus turns to the existence of ψ_v-successors for all relevant $\bar{c}$-states:

$$\bar{c} \wedge \mathcal{A}_v \wedge T \wedge \neg\mathcal{L}'_v \ . \tag{18}$$

If unsatisfiable, L_v is updated to $L_v \vee \bar{c}$, and generalization is complete.

Otherwise, a witness state s exists; it is checked for L_v-successors:

$$s \wedge T \wedge \mathcal{L}'_v \ . \tag{19}$$

If the query is satisfiable, then there exists s-successor state t and input j such that $t \models \mathcal{L}_v$ and $(s, j, t') \models T$. In this case, the following query is unsatisfiable:

$$s \wedge \mathcal{A}_v \wedge j \wedge T \wedge (\neg\mathcal{L}_u \vee \neg\mathcal{L}'_v) \ . \tag{20}$$

Its unsatisfiable core reveals a cube $\bar{s} \subseteq s$ with which to update L_v to $L_v \vee \bar{s}$, which eliminates s as a counterexample to query (18). If query (19) is unsatisfiable, then s is a CTG to be handled as described for EX nodes.

3.3 Refinements

Two refinements are immediate. First, to detect early termination, each time some node u's L_u or U_u is updated, its parent v is notified, and the proper update is made to its L_v and U_v, as explained in Section 3.1. If there is a (semantic) change in at least one of L_v and U_v, then the upward propagation continues. If the root node is modified so that a termination criterion is met ($I \wedge \neg U_0$ is satisfiable or $I \wedge \neg L_0$ is unsatisfiable), then the proof is complete.

[4] If v is an EU-node, notice that the term $\mathcal{A}_v$ in the antecedent excludes considering $\mathcal{L}_w$-states of $\bar{c}$, for such states are already decided for node v.

Consider the property of Example 1. If it fails, there is at least one trace leading from an initial state to a state s that falsifies $\mathsf{EF}\, p$. The outer EF node would direct IICTL to find such a trace, after which the upper bound query of the inner EF-node would return a proof that s cannot reach a p-state. As soon as the proof is generated, it is evident that the property is false.

Second, in Section 3.1, individual task states are submitted to nodes. However, multi-state initial conditions and counterexample traces create sets of task states. While generalization mitigates the cost of handling each state of a task set individually, even better is to allow nodes to reason about multi-state tasks. To do so, a node of the DAG receives a task set $\mathcal{T}$ with an associated label. The label All indicates that all states $t \in \mathcal{T}$ must satisfy ψ_v, while the label One indicates that at least one state $t \in \mathcal{T}$ must satisfy ψ_v.

Now function decide takes three arguments: $\mathsf{decide}(\mathcal{T},\ \ell,\ v)$, where $\ell \in \{\mathsf{All}, \mathsf{One}\}$. Initially, $\mathsf{decide}(I,\ \mathsf{All},\ 0)$ is called. A general invocation $\mathsf{decide}(\mathcal{T},\ \ell,\ v)$ immediately returns false if $\ell = \mathsf{All}$ and $\mathcal{T} \wedge \neg U_v$ is satisfiable, or if $\ell = \mathsf{One}$ and $\mathcal{T} \wedge U_v$ is unsatisfiable; and returns true if $\ell = \mathsf{One}$ and $\mathcal{T} \wedge \mathcal{L}_v$ is satisfiable, or if $\ell = \mathsf{All}$ and $\mathcal{T} \wedge \neg \mathcal{L}_v$ is unsatisfiable. Otherwise, it updates $\mathcal{T}$ to $\mathcal{T} \wedge \mathcal{A}_v$—that is, the undecided subset of $\mathcal{T}$—and continues.

If v is a $\neg$-node, then it switches the label and passes the task onto its child. If v is a $\psi_u \wedge \psi_w$-node and $\ell = \mathsf{All}$, then $\mathsf{decide}(\mathcal{T},\ \mathsf{All},\ u)$ is called. A return value of false indicates that some state $t \in \mathcal{T}$ falsifies ψ_u, so this call returns false as well. Otherwise, $\mathsf{decide}(\mathcal{T},\ \mathsf{All},\ w)$ is called and its return value returned.

If $\ell = \mathsf{One}$, the situation is more interesting: a state $t \in \mathcal{T}$ must be identified that satisfies both ψ_u and ψ_w. Therefore, $\mathsf{decide}(\mathcal{T},\ \mathsf{One},\ u)$ is called. A return value of false indicates that all states of $\mathcal{T}$ falsify ψ_u, so this call returns false, too. However, a return value of true indicates that at least one state of $\mathcal{T}$ satisfies ψ_u, and these states are now included in L_u. Therefore, $\mathsf{decide}(\mathcal{T} \wedge \mathcal{L}_u,\ \mathsf{One},\ w)$ is called to see if any of the identified states also satisfies ψ_w. If so, this call returns true. If not, then v's new bounds exclude some states of $\mathcal{T}$, including the ones that were found to satisfy ψ_u. $\mathcal{T}$ is consequently set to $\mathcal{T} \wedge U_v \wedge \neg L_u$, and the process is iterated.

If v is an EX-, EU-, or EG-node and $\ell = \mathsf{One}$, then its queries are executed iteratively with $\mathcal{T} \wedge U_v$ as the source states until either $\mathcal{T} \wedge \mathcal{L}_v$ is satisfiable, in which case true is returned, or $\mathcal{T} \wedge U_v$ is unsatisfiable, in which case false is returned. If $\ell = \mathsf{All}$, then v's queries are executed iteratively with $\mathcal{T} \wedge \neg \mathcal{L}_v$ as the source states until either $\mathcal{T} \wedge \neg U_v$ becomes satisfiable, in which case false is returned, or $\mathcal{T} \wedge \neg \mathcal{L}_v$ becomes unsatisfiable, in which case true is returned. The handling of multiple initial states by the reach and fair queries themselves is the main advantage of the multi-state refinement of decide.

With the handling of multi-state tasks defined, it remains to define how to create such tasks. Suppose during analysis of node v, where $\psi_v = \mathsf{E}\, \psi_u\, \mathsf{U}\, \psi_w$, decide discovers a trace $s_0, \ldots, s_n$ in which it must be decided whether states $s_0, \ldots, s_{n-1}$ are ψ_u-states and state s_n is a ψ_w-state. Then $\mathsf{decide}(\{s_0, \ldots, s_{n-1}\},\ \mathsf{All},\ u)$ and $\mathsf{decide}(\{s_n\},\ \mathsf{All},\ w)$ are called. Similarly, if $\psi_v = \mathsf{EG}\, \psi_u$, the states of a purported fair cycle $s_0, \ldots, s_n$ are decided with $\mathsf{decide}(\{s_0, \ldots, s_n\},\ \mathsf{All},\ u)$.

3.4 Fairness

Fairness in CTL cannot be handled completely within the logic itself. Instead, model checkers must be able to handle fairness constraints algorithmically when deciding whether a state satisfies an EG formula, a task that IICTL accomplishes by passing the constraints to fair. To show that finite paths computed for other types of formulae can be extended to fair paths, it suffices to show that they end in states that satisfy $\mathsf{EG}\,\top$. Hence, it is customary in BDD-based CTL model checkers to pre-compute the states that satisfy $\mathsf{EG}\,\top$ and constrain the targets of EU and EX computations to them [12].

IICTL instead tries to decide the fairness of as few states as possible. To that effect, it computes from the given φ a modified formula $\tau(\varphi)$ recursively defined as follows:

$$\begin{aligned} \tau(p) &= p & \tau(\mathsf{EG}\,\varphi) &= \mathsf{EG}\,\tau(\varphi) \\ \tau(\neg\varphi) &= \neg\tau(\varphi) & \tau(\mathsf{EX}\,\varphi) &= \mathsf{EX}(\tau(\varphi) \wedge \psi) \\ \tau(\varphi_1 \wedge \varphi_2) &= \tau(\varphi_1) \wedge \tau(\varphi_2) & \tau(\mathsf{E}\,\varphi_1\,\mathsf{U}\,\varphi_2) &= \mathsf{E}\,\tau(\varphi_1)\,\mathsf{U}(\tau(\varphi_2) \wedge \psi)\ , \end{aligned}$$

where

- p is an atomic proposition, and
- $\psi = \top$ if φ is a positive Boolean combination of EX, EU and EG formulae; $\psi = \mathsf{EG}\,\top$ otherwise.

For example, $\tau(\mathsf{AG}\,\mathsf{EF}(p \wedge \neg q)) = \tau(\neg\,\mathsf{EF}\,\neg\,\mathsf{EF}(p \wedge \neg q)) = \neg\,\mathsf{EF}(\neg\,\mathsf{EF}((p \wedge \neg q) \wedge \mathsf{EG}\,\top) \wedge \mathsf{EG}\,\top)$, while $\tau(\mathsf{AG}\,\mathsf{AF}\,p) = \tau(\neg\,\mathsf{EF}\,\mathsf{EG}\,\neg p) = \neg\,\mathsf{EF}\,\mathsf{EG}\,\neg p$.

While the definition of $\tau(\varphi)$ is closely related to the one implicitly used by most BDD-based model checkers—the difference is that in the latter, ψ always equals $\mathsf{EG}\,\top$—it minimizes checks for fairness by taking into account that every path with a fair suffix is fair.

For instance, in the case of $\mathsf{AG}\,\mathsf{AF}\,p$, IICTL does not check whether any state satisfies $\mathsf{EG}\,\top$ because the states that satisfy $\mathsf{EG}\,p$ are known to be fair. For the resetability property $\mathsf{AG}\,\mathsf{EF}(p \wedge \neg q)$, however, a state that satisfies $p \wedge \neg q$ is not assumed to satisfy the inner EF node unless it is proved fair.

4 Results

The IICTL algorithm has been implemented in the IImc model checker [11], and it has been evaluated on a set of 33 models (mostly from [18]) for a total of 363 CTL properties (278 passing and 85 failing). These properties include only one invariant, since IICTL delegates invariant checking to IC3. No collection of branching time properties used in real designs similar to that available for safety properties is available. This experimental evaluation is therefore preliminary. The experiments have been run on machines with 2.8 GHz Intel Core i7 CPUs and 9 GB of RAM. A timeout of 300 s was imposed on all runs.

In this section the performance of IICTL is compared to that of the BDD-based CTL model checker in VIS-2.3 [18] (with and without preliminary reachability analysis) and, for the properties that can be expressed in LTL, to the FAIR and IC3 algorithms [4, 6]. The total run times were: 27613 s for IICTL; 32220 and 31555 s for VIS with and without reachability analysis.

Table 2 shows for each of the three CTL algorithms the numbers of timeouts (TO) and the numbers of properties that could be solved by only one technique (US). Only the models for which timeouts occurred are listed. While IICTL

Table 2. Timeouts and Unique Solves (nr = no reachability)

model	IICTL		VIS		VIS-nr	
	TO	US	TO	US	TO	US
am2901		1	2		1	
am2910			3			
CAB	9			9	11	
checkers	46	6	52		52	
cube			1			
gcd	2					
newnim	4					
palu			5			
redCAB	5			5	8	
rether	1			1	1	

model	IICTL		VIS		VIS-nr	
	TO	US	TO	US	TO	US
rgraph	1					
tarb16		16	16		16	
soap	10		10		10	
swap	2					
vcordic	1		1		1	
viper	1		3			
vsa16a	2		2		1	1
vsaR			2			
vsyst	1		1		1	
total	85	23	98	15	102	1

obtains the lowest number of timeouts and the highest number of unique solutions, it is apparent that the three methods have different strengths and thus are complementary. This point is further brought out by the plots of Figure 3.

The upper row of Figure 3 shows the comparison of IICTL to VIS in the form of scatterplots. The lower left plot compares IICTL to FAIR and IC3 for 110 properties expressible in both CTL and LTL. IC3 is used for safety properties, and FAIR is used for the other ones. Finally, the lower right plot compares the best of IICTL and IC3/FAIR to the best result obtained by VIS, with or without reachability. Even with the averaging effect of taking the best results between two methods, there remain significant differences between the incremental, inductive approach and the one based on BDDs.

The data shown for IICTL were obtained with a medium level of effort in trace generalization (option (2) in Section 3.2). This approach proved the most robust and time-effective, though occasionally, the highest level of effort pays off. This is more likely to be the case when precise reachability information is crucial (e.g., with *rether*, which has very few reachable states).

The comparison of IICTL to the automata-based approach that uses IC3 or FAIR as decision procedure shows that IICTL is close in performance to techniques that are specialized for one type of property. There are three properties for which IICTL times out but IC3 does not. (They are all safety properties.) IICTL gets mired in difficult global reachability queries, because the current

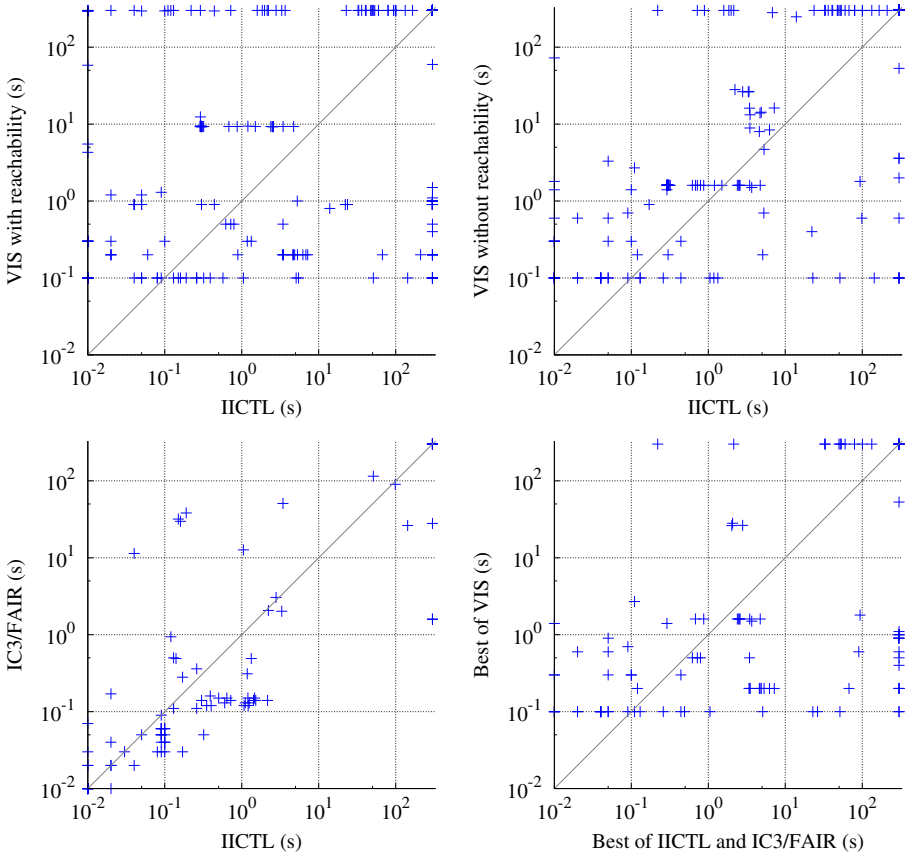

Fig. 3. Comparing IICTL to competing techniques

implementation does not know that the set of target states will eventually prove empty. On the the other hand, since the properties are easily strengthened to inductive, IC3 terminates quickly. These comparisons highlight areas in which progress should be made by making IICTL's strategy more flexible and nuanced.

5 Conclusion

Building on the ideas of incremental, inductive verification (IIV) pioneered in IC3 and FAIR, IICTL is a new property-directed abstracting model checker for CTL with fairness. Although the implementation is preliminary, the experimental results show that it is competitive in robustness and, importantly, complementary to the traditional symbolic BDD-based algorithm. IICTL offers a different approach to model checking that allows it to prove some properties on systems for which, like `checkers`, BDDs are unwieldy. The techniques for handling CTL properties in an IIV style—the task-based algorithm structured around the parse

graph of the CTL property, and forall-exists generalization of traces—will contribute to the next goal for IIV: CTL* model checking.

References

[1] Abdulla, P.A., Bjesse, P., Eén, N.: Symbolic Reachability Analysis Based on SAT-Solvers. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 411–425. Springer, Heidelberg (2000)

[2] Bhat, G., Cleaveland, R., Grumberg, O.: Efficient on-the-fly model checking for CTL*. In: LICS, pp. 388–397 (June 1995)

[3] Bradley, A.R.: k-step relative inductive generalization. Technical report, CU Boulder (March 2010), http://arxiv.org/abs/1003.3649

[4] Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)

[5] Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD, pp. 173–180 (November 2007)

[6] Bradley, A.R., Somenzi, F., Hassan, Z., Zhang, Y.: An incremental approach to model checking progress properties. In: FMCAD, pp. 144–153 (November 2011)

[7] Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: FMCAD, pp. 135–143 (November 2011)

[8] Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)

[9] Du, X., Smolka, S.A., Cleaveland, R.: Local model checking and protocol analysis. STTT 3(1), 219–241 (1999)

[10] Larsen, K.G.: Proof systems for Hennessy-Milner logic with recursion. TCS 72(2-3), 265–288 (1990)

[11] http://iimc.colorado.edu

[12] McMillan, K.L.: Symbolic Model Checking. Kluwer, Boston (1994)

[13] McMillan, K.L.: Applying SAT Methods in Unbounded Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)

[14] Oshman, R.: Bounded model-checking for branching-time logic. Master's thesis, Technion, Haifa, Israel (June 2008)

[15] Quielle, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)

[16] Ravi, K., Somenzi, F.: Minimal Assignments for Bounded Model Checking. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 31–45. Springer, Heidelberg (2004)

[17] Stirling, C., Walker, D.: Local model checking in the modal μ-calculus. TCS 89(1), 161–177 (1991)

[18] http://vlsi.colorado.edu/~vis

[19] Wang, B.-Y.: Proving $\forall\mu$-Calculus Properties with SAT-Based Model Checking. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 113–127. Springer, Heidelberg (2005)

[20] Woźna, B.: ACTL* properties and bounded model checking. Fundamenta Informaticae 63(1), 65–87 (2004)