An Incremental Approach to Model Checking Progress Properties

Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, Yan Zhang
Dept. of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder
Email: bradleya@colorado.edu, fabio@colorado.edu

***Abstract*—An incremental algorithm for model checking progress properties is proposed. It follows from the following insight: any SCC-closed region of a system's state graph can be represented by a sequence of inductive assertions. Each iteration of the algorithm selects a set of states, called a skeleton, that together satisfy all fairness conditions; it then applies safety model checkers to attempt to connect the states into a reachable fair cycle. If this attempt fails, the resulting learned lemma takes one of two forms: an inductive reachability assertion that shows that at least one state of the skeleton is unreachable, or an inductive wall that defines two SCC-closed regions of the state graph. Subsequent skeletons must be chosen entirely from one side of the wall. Because a lemma often applies more generally than to the one skeleton from which it was derived, property-directed abstraction is achieved. The algorithm is highly parallelizable.**

I. Introduction

An incremental-style analysis, one that generates many intermediate lemmas on the way to a proof, yields property-focused abstraction, speed, and the possibility of parallelism. IC3 demonstrated the power of incrementality for safety model checking [1]. In this paper, we introduce an incremental algorithm for model checking progress properties [2] that harnesses safety model checkers.

While alternatives exist for lifting safety model checkers to progress properties [3], incrementality in itself is a worthwhile goal—whether one is using parallel resources to implement a portfolio of many safety model checkers [4] or applying the resources to accelerate computation [1]. In addition to using computational resources well, an incremental-style model checker generalizes from specific cases of why the property might not hold to intermediate lemmas about aspects of the system that are relevant to proving the property. In this way, it achieves property-focused abstraction of the system, and like a human verifier, it invests relatively little computation into discovering each lemma.

An *SCC-closed* region of the state graph is such that every SCC (*strongly connected component*) is either entirely contained in the region or entirely disjoint from the region. The fundamental insight for making an incremental progress model checker is that any SCC-closed region of a system's state graph can be represented by a sequence of inductive assertions. In other words, intermediate lemmas to characterize the SCCs of the state graph can take the form of inductive assertions. Each assertion defines a *one-way wall* that transects the state SCC graph. Given a selection of states, called a *skeleton*, that together satisfy the fairness conditions, one can prove via safety queries that (1) at least one of the states is unreachable from the system's initial condition; that (2) one of the skeleton states cannot reach another, providing a one-way wall; or that (3) the skeleton can actually be completed to form a "lasso"-shaped counterexample. How to use the second outcome is the crux of the algorithm.

Suppose that P is the one-way wall, an inductive proof that one skeleton state cannot reach another. Any fair cycle must occur completely on one side of the wall: all of its states must either satisfy P, or they must all satisfy $\neg P$. For once a path crosses the wall, it cannot return. Hence, when finding fair cycles, the transition relation can be strengthened by the constraint $P \leftrightarrow P'$, which excludes transitions that cross the wall P. (Technically, because P is inductive, only $\neg P \rightarrow \neg P'$ is necessary.) This constraint is the incremental information expressed by the lemma P. Subsequent skeletons must be chosen from one side of the wall or the other, and eventually every reachable *arena* defined by the sequence of walls must become unfair, if the progress property indeed holds. A crucial characteristic of a proof, when IC3 is used as the safety model checker, is that it potentially splits many arenas, not just the arena from which the skeleton was selected. Hence, not every arena need be examined explicitly.

After introducing the problem domain (Section II), Section III describes the algorithm in detail. Then Section IV relates the proposed algorithm to previous work. Finally, Section V investigates empirical characteristics of the algorithm in relation to other well-known techniques.

II. Background

Following standard practice, we represent a *finite-state system* as a tuple $S : (\bar{i},\ \bar{x},\ I(\bar{x}),\ T(\bar{i},\bar{x},\bar{x}'))$ consisting of primary inputs $\bar{i}$, state variables $\bar{x}$, a propositional formula $I(\bar{x})$ describing the initial configurations of the system, and a propositional formula $T(\bar{i},\bar{x},\bar{x}')$ describing the transition relation. Primed state variables $\bar{x}'$ represent the next state.

A state of the system is an assignment of Boolean values to all variables $\bar{x}$ and is described by a *cube* over $\bar{x}$, 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 $s \models F$, we say that s is an F-*state*. A formula F *implies* another formula G, written $F \Rightarrow G$, if every satisfying assignment of F satisfies G.

A *clause* is a disjunction of literals. A subclause $d \subseteq c$ is a clause d whose literals are a subset of c's literals.

A *run* of S, $s_0, s_1, s_2, \ldots$, which may be finite or infinite in length, is a sequence of states such that $s_0 \models I$ and for each adjacent pair (s_i, s_{i+1}) in the sequence, $\exists \overline{i}.(\overline{i}, s_i, s'_{i+1}) \models T$. That is, a run is the sequence of assignments in an execution of the transition system. A state that appears in some run of the system is *reachable*.

An *invariance property* $P(\overline{x})$, a propositional formula, asserts that only P-states are reachable. P is *invariant* for the system S (that is, S-invariant) if indeed only P-states are reachable. If P is not invariant, then there exists a finite *counterexample* run $s_0, s_1, \ldots, s_k$ such that $s_k \not\models P$. An invariance property $P(\overline{x})$ is *inductive* if

1) (*initiation*) every initial state satisfies the property: $I(\overline{x}) \Rightarrow P(\overline{x})$; and
2) (*consecution*) every transition from a P-state leads to a P-state: $P(\overline{x}) \wedge T(\overline{i}, \overline{x}, \overline{x}') \Rightarrow P(\overline{x}')$.

An assertion F is *inductive relative to* another assertion G, possibly containing primed variables, if

1) every initial state satisfies F: $I(\overline{x}) \Rightarrow F(\overline{x})$; and
2) F satisfies consecution under assumption G: $G(\overline{x}, \overline{x}') \wedge F(\overline{x}) \wedge T(\overline{i}, \overline{x}, \overline{x}') \Rightarrow F(\overline{x}')$.

Relative induction is useful for gaining knowledge about a system in an incremental fashion [2].

Checking a *safety property* of S is reducible to checking an invariance property. While the work described in this paper makes use of safety model checkers, the primary focus is on analyzing *progress properties* [2]. For this purpose, we need to introduce *fairness* into our system models. A *Büchi fairness condition* $B(\overline{x})$ of a system S is a propositional formula that constrains the infinite runs of S: infinite run $s_0, s_1, s_2, \ldots$ is a *computation* of S if infinitely many s_i satisfy B, $s_i \models B$. We represent a system with fairness conditions as the augmented tuple $S : (\overline{i},\ \overline{x},\ I(\overline{x}),\ T(\overline{i}, \overline{x}, \overline{x}'),\ \mathcal{B} : \{B_1(\overline{x}), \ldots, B_\ell(\overline{x})\})$. The fundamental question that this paper addresses is that of *language emptiness*: *Does S lack computations?*

Model checking LTL properties of systems motivates this problem. Deciding whether a system S satisfies LTL property P is reducible to checking language emptiness of the system constructed as the parallel composition of S and the *Büchi automaton A* for $\neg P$. The resulting system inherits the fairness conditions of S as well as one additional fairness condition, the Büchi acceptance condition of A.

A fairness condition B of S is *weak* [5] if for every computation of S there exists k such that $i \geq k \Rightarrow s_i \models B$. Weak fairness conditions correspond to *persistence properties* [2]. Multiple weak conditions can be reduced to just one weak condition so that the search for fair cycles can be restricted to the reachable states that lie on some cycle where the weak condition holds globally. When a fairness condition of S is inherited from a Büchi automaton, its strength is at most the strength of the fairness condition of the automaton [6].

Because S is finite-state, a nonempty language described by system S with fairness conditions must have a computation that takes the form of a *reachable fair cycle*: a "lasso" consisting of a "stem" (a finite run) from an initial state s_0, $s_0 \models I$, to an intermediate state s_i, and a "loop" (also a finite run) from s_i back to itself that contains at least one state s_j satisfying each fairness condition B_j. Our algorithm searches for such reachable fair cycles.

III. Fair: An Incremental Algorithm

A. The Basic Algorithm

The algorithm works in the following manner. It iteratively executes a *skeleton query* to obtain a set of states that together satisfy all fairness conditions. If the query is ever unsatisfiable, the algorithm concludes that the language of S is empty. It next attempts to complete the skeleton into a reachable fair cycle by executing a set of safety model checking queries to connect the initial states to one state of the skeleton, and each state of the skeleton to another in such a way as to create a cycle. If it succeeds, then it has found a reachable fair cycle and thus concludes that the language of S is not empty. Otherwise, one of the safety queries fails and returns an inductive proof. If the *stem query*, which attempts to connect an initial state to a skeleton state, fails, then the proof provides new global unreachability information. If a *cycle query*, which attempts to connect one skeleton state to another, fails, then the proof yields new information about the SCC structure of S. In particular, the proof says that a fair cycle, if one exists, must occur completely on one side or the other of the inductive proof (that is, all states of the cycle must satisfy the proof, or all states must falsify it). Both situations thus cause the algorithm to make progress, so that it eventually must find a reachable fair cycle or conclude that one does not exist.

In detail, consider system $S : (\overline{i},\ \overline{x},\ I(\overline{x}),\ T(\overline{i}, \overline{x}, \overline{x}'),\ \mathcal{B} : \{B_1, \ldots, B_\ell\})$. Let $\mathcal{R}$ denote a growing list of global reachability assertions, each of which is inductive relative to its predecessors and provides information about unreachable states. Let $\mathcal{W}$ denote a growing list of *walls* that no fair cycle can cross, each of which satisfies consecution relative to previously generated walls and $\mathcal{R}$, as discussed in detail later. Walls represent learned information about the SCC structure of the state graph of S. A set of walls $\mathcal{W}$ defines $2^{|\mathcal{W}|}$ (possibly empty) arenas; each arena (and, consequently, any union of arenas) is SCC-closed. Both lists are empty initially.

The *skeleton query* returns a *skeleton* or, if unsatisfiable, indicates that the language of S is empty. A skeleton consists of a set of states that together satisfy all fairness conditions. The query requires (in its complete form, but see Section III-B) one copy of $\overline{x}$ for each fairness condition $B \in \mathcal{B}$:

$$\bigwedge_{B \in \mathcal{B}} \left[\begin{array}{l} \qquad B(\overline{x}_B) \wedge \bigwedge_{R \in \mathcal{R}} R(\overline{x}_B) \\ \wedge \quad \bigwedge_{W \in \mathcal{W}} (c_W \rightarrow W(\overline{x}_B)) \wedge (\neg c_W \rightarrow \neg W(\overline{x}_B)) \end{array} \right]$$

The first line of the query requires that the states of a model be such that each fairness condition is represented by states not known to be unreachable. The second line requires that all states of a model come from the same *arena*, that is, are on the same side of each wall $W \in \mathcal{W}$. The *choice variables* c_W for $W \in \mathcal{W}$ achieve this requirement: a model can only have one assignment to each choice variable, and that assignment determines on which side of each wall the skeleton appears.

If the skeleton query is satisfiable, any model describes some set of states $\{s_0, \ldots, s_{n-1}\}$, where $n \leq |\mathcal{B}|$, that satisfy the fairness conditions, that are not known to be unreachable, and that are not separated by any previously discovered wall ($n < |\mathcal{B}|$ if some state satisfies multiple fairness conditions and appears more than once). The task, then, is to attempt to complete the skeleton into a reachable fair cycle or, if the attempt fails, to learn new information in the form of an inductive reach assertion or a wall about why any reachable cycle of S cannot contain all of the states of the skeleton.

Any safety model checker that produces counterexample runs or inductive proofs can address this task, although we discuss later why proofs from certain model checkers, like IC3, make better walls. 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 inductive proof $P(\overline{x})$ separating F from G, that is, such that

- $F(\overline{x}) \Rightarrow P(\overline{x})$,
- $P(\overline{x}) \Rightarrow \neg G(\overline{x})$, and
- $C(\overline{x}, \overline{x}') \wedge P(\overline{x}) \wedge T(\overline{i}, \overline{x}, \overline{x}') \Rightarrow P(\overline{x}')$.

Notice that P is inductive relative to the constraints C.

For an n-state skeleton, $n + 1$ reach-queries are required. One *stem query* determines if the skeleton is reachable, given the learned reachability information $\mathcal{R}$:

$$\mathsf{reach}\left(S,\ \bigwedge_{R \in \mathcal{R}} R(\overline{x}),\ I,\ s_0\right)\ . \tag{1}$$

This query asks whether s_0 is reachable from an I-state. While the previous reachability information is not necessary, it is provided to restrict the search. One could instead pose the more general query in which the disjunction of all skeleton states, $\bigvee_{i=0}^{n-1} s_i$, is the target; or pose n queries, one for each state s_i, depending on computational resources. If an instance of a stem query is unsatisfiable, the proof is added to $\mathcal{R}$.

The remaining n queries are *cycle queries*, which determine if each state s_i can reach a successor $s_{i \oplus_n 1}$, where $\oplus_n$ is addition modulo n. One can pose up to n^2 queries if the computational resources are available. These queries are more complicated than the stem query because more previously-derived information can be used.

A naive cycle query takes the following form:

$$\mathsf{reach}\left(S,\ \mathsf{true},\ s_i,\ s_{i \oplus_n 1}\right)\ . \tag{2}$$

If s_i cannot reach $s_{i \oplus_n 1}$, then the query returns an inductive proof P: $s_i \Rightarrow P$, $P \wedge T \Rightarrow P'$, and $P \Rightarrow \neg s_{i \oplus_n 1}$. P is a wall: no cycle can cross it because no P-state has a $\neg P$-state successor. While a $\neg P$-state can have a P-state successor, crossing the wall is pointless when searching for a cycle since it cannot be crossed again. P can thus be added to $\mathcal{W}$, the list of walls that no fair cycle can cross.

However, this query does not exploit known information. For a cycle query, each wall $W \in \mathcal{W}$ constrains the transition relation as follows:

- If no W*-skeleton* (a skeleton whose states are W-states) exists, then $\neg W \wedge \neg W'$.
- If no $\neg W$-skeleton exists, then $W \wedge W'$.
- Otherwise (if both sides contain skeletons), $W \leftrightarrow W'$.

Unfortunately, encoding the full constraints in the cycle queries requires a quantifier alternation. Instead, each new wall W is tested to learn a new constraint on T; such constraints are collected in the *constraint list* $\mathcal{C}$:

- If no W-skeleton exists, then add $\neg W \wedge \neg W'$. (Technically, because W is inductive, $\neg W'$ is sufficient.)
- If no $\neg W$-skeleton exists, then add $W \wedge W'$. (Technically, W is sufficient.)
- Otherwise, add $W \leftrightarrow W'$. ($W' \rightarrow W$ is sufficient.)
- Optionally, if W is determined (heuristically) to be uninteresting for constraining T, do not add a constraint.

It is also possible to exclude regions defined by multiple walls—even individual arenas—that lack fair skeletons. However, this more general heuristic, while potentially useful at the beginning of the analysis, is too expensive for general use. The list $\mathcal{C}$ is used to constrain T during the cycle query:

$$\mathsf{reach}\left(S,\ \bigwedge_{R \in \mathcal{R}} R \wedge \bigwedge_{C \in \mathcal{C}} C,\ s_i,\ s_{i \oplus_n 1}\right)\ . \tag{3}$$

This query is satisfiable precisely when the naive cycle query (2) is satisfiable. However, a proof discovered during evaluating this query need only be inductive relative to the information contained in $\mathcal{R}$ and $\mathcal{C}$ rather than on its own.

There is one technicality: when there is only one state in the skeleton, the form of the single cycle query is different. A *single-state skeleton cycle query* determines if a state s_0 can reach itself nontrivially, which is stated as a query determining whether the successors of s_0 can reach s_0:

$$\mathsf{reach}\left(S,\ \bigwedge_{R \in \mathcal{R}} R \wedge \bigwedge_{C \in \mathcal{C}} C,\ \mathsf{post}(S, s_0),\ s_0\right)\ . \tag{4}$$

(Safety model checkers such as IC3 can be modified in such a way that the post-image does not have to be computed explicitly.) Additionally, a proof P does not eliminate the same skeleton from further consideration. P, as a wall, separates s_0 (which satisfies $\neg P$) from its successors (which satisfy P). However, s_0 can be selected as a skeleton again. There are several solutions to avoid this nontermination situation: (1) constrain the skeleton query so that only states with some successors in the same arena can be selected, (2) for a cycle proof P, construct a wall defined by $W = P$ and $\neg W = \neg P \wedge \neg s_0$ instead of the usual wall defined by $W = P$ and $\neg W = \neg P$. More powerful refinements of each of these solutions are discussed in Sections III-B and III-C.

If all queries (1) and (3/4) return counterexamples, the runs are assembled into a computation that takes the form of a lasso, proving that the language of S is nonempty. Otherwise, a proof P returned by the stem query provides new global reachability information, so P is added to $\mathcal{R}$; or a proof P returned by one of the cycle queries provides new information about the SCC structure of S, so P is added to the set of walls, $\mathcal{W}$, and a new constraint may be derived from P and added to $\mathcal{C}$. Then with this new information, the algorithm again executes the skeleton query. The new information is sufficient to exclude the same skeleton from being selected again.

Several aspects of this basic algorithm are nondeterministic and thus invite further detail and heuristics:

- Selection of the skeleton (Section III-B).
- The order in which the stem query and cycle queries are executed (Section III-F).
- The proofs themselves (Section III-D).
- Whether to derive a new constraint on T from a wall W. Technically, none are required for completeness; using some accelerates the search; and using all can slow the search. Our implementation derives a new constraint from W if one side of W lacks skeletons or if W consists of a single clause.

Section III-E discusses an incomplete but effective method of discovering information about the SCC structure independently of skeletons.

B. Choosing Skeletons

We discuss two enhancements to the basic algorithm. The first minimizes the number of states in skeletons by formulating the skeleton query to force states to satisfy multiple fairness conditions when possible. The intuition is that the discovered walls might explain more if the separated states satisfy multiple fairness conditions. The second enhancement adds constraints to the skeleton query to force a selected state to have at least K-step successor and predecessor sequences of different states within the arena, unless some state in these sequences is the state itself. This enhancement effectively reduces the number of skeletons to consider. For $K > 0$, single-state skeletons with no successors cannot be chosen, so that this enhancement addresses the termination issue raised in the previous section.

To (heuristically) minimize the number of states selected, let $j : \mathcal{B} \to \{1, \ldots, |\mathcal{B}|\}$ be a map from the Büchi fairness conditions of S to indices, where j can map different fairness conditions to the same index. The skeleton query then has the following form:

$$\bigwedge_{B\in\mathcal{B}} \left[\begin{array}{l} B(\overline{x}_{j(B)}) \wedge \bigwedge\limits_{R\in\mathcal{R}} R(\overline{x}_{j(B)}) \\ \wedge \bigwedge\limits_{W\in\mathcal{W}} (c_W \rightarrow W(\overline{x}_{j(B)})) \wedge (\neg c_W \rightarrow \neg W(\overline{x}_{j(B)})) \end{array} \right]$$

Potentially fewer copies of the assertions are required.

Of course, the query is only complete, in the sense that its unsatisfiability implies the emptiness of the language of S, when each condition is mapped to a unique index. Hence, the modified algorithm finds a map j that (heuristically) minimizes the number of unique indices while still producing a satisfiable query. If because of new information the query becomes unsatisfiable, a new map, which may have the same number of unique indices but must at least combine conditions differently, is generated. Only when the query corresponding to a bijective mapping is unsatisfiable does the algorithm conclude that the language of S is empty.

The second enhancement reduces the number of potential skeletons by requiring selected states to have nontrivial sequences of successors and predecessors. For each unique index, $2K$ unrollings of the transition system are asserted with time-steps ranging from $-K$ to K. An additional constraint asserts that either the predecessor sequence or the successor sequence includes $\overline{x}^0_{j(B)}$ itself, or otherwise that the predecessor sequence and the successor sequence are each loop-free, yielding the following skeleton query:

$$\bigwedge_{B\in\mathcal{B}} \left[\begin{array}{l} B(\overline{x}^0_{j(B)}) \wedge \bigwedge\limits_{R\in\mathcal{R}} R(\overline{x}^{-K}_{j(B)}) \\ \wedge \bigwedge\limits_{k\in\{-K,\ldots,K-1\}} T(\overline{i}^k_{j(B)}, \overline{x}^k_{j(B)}, \overline{x}^{k+1}_{j(B)}) \\ \wedge \left(\begin{array}{l} \bigvee\limits_{k\in\{-K,\ldots,-1,1,\ldots,K\}} \overline{x}^k_{j(b)} = \overline{x}^0_{j(b)} \\ \vee \quad (\mathsf{loopFree}_{<0} \wedge \mathsf{loopFree}_{>0}) \end{array} \right) \\ \wedge \bigwedge\limits_{W\in\mathcal{W}} (c_W \rightarrow W(\overline{x}^{-K}_{j(B)})) \wedge (\neg c_W \rightarrow \neg W(\overline{x}^{K}_{j(B)})) \end{array} \right]$$

where

$$\mathsf{loopFree}_{<0} = \bigwedge_{k\in\{-K,\ldots,-2\}} \bigwedge_{\ell\in\{k+1,\ldots,-1\}} \overline{x}^k_{j(B)} \neq \overline{x}^\ell_{j(B)} ,$$

and $\mathsf{loopFree}_{>0}$ is similarly defined.

C. Single-State Skeletons

Recall that single-state skeletons must be handled via a single-state cycle query (4) that determines whether the successors of s_0 can reach s_0. If the query returns a proof P, then the successors of s_0 must satisfy P since they define the initial condition, while s_0 itself must falsify P since it defines the target. In other words, the proof P cuts the state space directly through the transitions between s_0 and its successors, so that s_0 is on the edge of an arena.

Consequently, the following propositional query, which asks if s_0 has any $\neg P$-successor, must be unsatisfiable:

$$s_0 \wedge \neg P \wedge T \wedge \bigwedge_{C\in\mathcal{C}} C \wedge \bigwedge_{R\in\mathcal{R}} R \wedge \neg P' \ .$$

From the unsatisfiable core one can extract a cube $d \subseteq s_0$ whose $\neg P$-states lack $\neg P$-state successors. Its negation can be conjoined to $\neg P$ to form one side of the wall: $\neg P \wedge \neg d$, which eliminates at least s_0 from consideration.

If a successor state t of s_0 is known, for example, when $K > 0$ (Section III-B), a similar query can test whether t has P-predecessors. If not, one can extract a cube $d \subseteq t$ from the core and strengthen P with $\neg d$.

D. Refining IC3 Proofs

IC3 discovers inductive proofs P in CNF [1]. While adequate as certificates of unreachability, which is what matters in the context of safety model checking, the proofs can be unnecessarily large and specific to the query. For example, a proof from a cycle query contains the clause $\neg s_{i \oplus_n 1}$. We describe several methods of manipulating an IC3 proof to make it more general and to reduce its size.

The property can be generalized. Let $P = F \wedge \neg s_{i \oplus_n 1}$. The MIC algorithm [7] is applied to $\neg s_{i \oplus_n 1}$ in the context of P to derive a subclause $c \subseteq \neg s_{i \oplus_n 1}$, yielding proof $\overline{P} = F \wedge c$.

The proof P can be strengthened by applying MIC iteratively to the clauses of P until no further changes are possible. We apply this manipulation to global reachability proofs.

The proof P can be weakened. Again, let $P = F \wedge c$, where $c \subseteq \neg s_{i \oplus_n 1}$. A MIC-like algorithm is applied to drop clauses of F. First, observe that one can use the unsatisfiable core of $F \wedge c \wedge T \wedge \neg P'$, corresponding to consecution, to reduce P: any clause of F that is not in the core is unnecessary. Second, observe that dropping an arbitrary clause d can result in a non-inductive assertion because d might be required to support other clauses. In this case, consecution fails with some counterexample states (t, t'). The set of clauses that t' falsifies in the next state must then be dropped, as they are no longer supported. Dropping these clauses may in turn require dropping other clauses, and so on. If ever c becomes unsupported (that is t' falsifies c'), the process must backtrack to the last inductive assertion; there, the same steps can be applied to a different clause unless all options have been explored. Alternately, if the process converges on an assertion for which consecution holds, the first observation can be used to further reduce the clause set. Then the clause-dropping process can be attempted again.

These manipulations can be combined to heuristically derive a minimally-sized proof: iteratively apply strengthening followed by weakening until no further changes can be made. Strengthening may reduce the number of literals, while weakening may reduce the number of clauses.

E. Skeleton-Independent Proofs

Skeletons serve to direct the exploration of the SCC structure of S; however, some important facts are not easily derived by this property-directed method.

Consider, for example, a system consisting of a single n-bit counter whose bits are named $b_0, \ldots, b_{n-1}$, where b_{n-1} is most significant; an output bit o that switches to 1 the first time that the counter reaches all 1s and then stays at 1; a fairness condition that asserts that infinitely often $\neg o$; and an initial condition in which all bits are 0. The system is unfair because $o = 0$ only for the first iteration through the counter's values. An ideal proof is constructed as follows:

- Inductive assertion o, since once o becomes 1, it stays 1. No skeleton exists among the o-states, so $\neg o \wedge \neg o'$ constrains future cycle queries.
- Inductive (relative to $\neg o$) assertion b_{n-1}, since once b_{n-1} becomes 1, it stays 1 in the $\neg o$ arena. Both sides of the proof have skeletons, so $b_{n-1} \leftrightarrow b'_{n-1}$ constrains future cycle queries.
- Inductive (relative to previous walls) assertion b_{n-2}, since once b_{n-2} becomes 1, it stays 1 in *every arena defined by the previous two proofs*.
- Similarly, inductive assertions $b_{n-3}, \ldots, b_0$ are derived in that order, each holding relative to prior information.

When $K > 0$ (see Section III-B), the skeleton query becomes unsatisfiable after these walls are generated: because of the learned constraint $b_0 \leftrightarrow b'_0$, each arena has only one state, and that state lacks a successor in its arena. The size of the proof is thus linear in the size of the counter. This proof sequence discovers the obvious ranking function.

Unfortunately, discovering the first fact with skeletons requires stumbling fortuitously upon the skeleton in which all $b_i = 1$ and $o = 0$. This state's only successor is the state in which all $b_i = 0$ and $o = 1$, so an inductive separating wall is indeed o. However, no other fair state has a successor in which $o = 1$, so the resulting walls cannot simply be o (since the successor must satisfy it) or $\neg o$ (since both the fair state and its successor satisfy it and thus are not separated). In other words, their walls must involve b_i literals and be less informative as a result. Discovering subsequent facts via skeletons requires similarly, although decreasingly, fortuitous selections; for example, to discover b_{n-1} requires examining precisely the one state for which $b_{n-1} = 0$ and whose successor has $b_{n-1} = 1$.

In contrast, iteratively testing whether any literal of the state variables of the system is itself a proof (that is, satisfies consecution relative to known information) produces the linear-sized proof quickly. Let ℓ be such a literal. Then if the formula

$$\bigwedge_{R \in \mathcal{R}} R \wedge \bigwedge_{C \in \mathcal{C}} C \wedge T \wedge \ell \wedge \neg \ell' \qquad (5)$$

is unsatisfiable, ℓ obeys consecution: once ℓ is true, it is true henceforth. In this case, ℓ is a wall.

While this heuristic is incomplete, its effectiveness on counters suggests that such simple queries should be executed frequently, for example, after each addition to $\mathcal{R}$ or $\mathcal{C}$. Experiments show that on more complicated systems, several iterations of skeleton-based wall construction create opportunities to learn new non-skeleton-directed proofs.

In addition to counters, this technique quickly derives information about property automata for favorable encodings of the automata's transition relations. A one-hot encoding, for example, reveals structural information readily. Predicates derived from the system description may also be effective candidates for this heuristic.

F. Executing Queries

The ideal computational environment in which to run this algorithm is a highly parallel one:

- $n + 1$ queries must be analyzed until either all yield counterexample runs or one yields a proof.
- Each query can be analyzed by a portfolio of safety model checkers, even incomplete methods: based on BDDs [8],

BMC [9], interpolation [10], IC3 [1], and simulation. While any counterexample run is informative, only proofs that are inductive are useful. However, proofs produced by non-approximating safety checkers (e.g., BDD-based) will cause fair to derive walls that are only useful in the arena from which the skeleton was drawn, thus hindering the algorithm's ability to generalize from skeletons, a key characteristic. Hence, we rely on IC3 for proofs.

- IC3 is itself parallelizable.
- As the overall methodology is incremental, multiple skeletons can be analyzed simultaneously in the same way that multiple counterexamples to induction can be analyzed simultaneously in IC3.

However, if parallel resources are unavailable, one observation has become clear from experimentation: queries must be analyzed in a time sharing fashion. Since only one query need be unsatisfiable to rule out a skeleton, a poor time allocation can cause excessive time to be wasted on finding irrelevant counterexample runs. Varying the order in which queries are executed also seems important, so that one fairness condition is not favored over others or over the stem query.

G. A Summary of the Algorithm

Figure 1 lists pseudocode for the fair algorithm.

```
1   bool fair($S$ : system, $K$ : uint):
2     $\mathcal{R}$ := $\emptyset$, $\mathcal{W}$ := $\emptyset$, $\mathcal{C}$ := $\emptyset$
3     {for full skeleton query}
4     $\iota$ := bijection between $\mathcal{B}$ and $\{1,\ldots,|\mathcal{B}|\}$
5
6     while skelQ($\mathcal{R}$, $\mathcal{W}$, $\iota$, $K$) is sat:
7       {for skeleton-minimization (§B)}
8       $j$ := map($\mathcal{R}$, $\mathcal{W}$, $K$)
9
10      while skelQ($\mathcal{R}$, $\mathcal{W}$, $j$, $K$) is sat:
11        result :=
12          heuristically choose:
13            {skeleton-independent proof (§E)}
14            let $\ell$ be a literal or other predicate
15              such that query (5) is unsat
16            $P$ := $\ell$
17          alternately:
18            {skeleton-based analysis (§A)}
19            $s_0,\ldots,s_{m-1}$ := skelQ($\mathcal{R}$, $\mathcal{W}$, $j$, $K$)
20            in parallel do until
21                all yield counterexamples
22                or one returns a proof:
23              stemQ($\mathcal{R}$, $s_0$)
24              for $i \in \{0,\ldots,m-1\}$:
25                cycleQ($\mathcal{R}$, $\mathcal{C}$, $s_i$, $s_{i\oplus_m 1}$)
26
27        if result is all counterexamples:
28          return true {non-empty language}
29
30        elif result is a proof $P$ from stemQ:
31          {new reachability information}
32          $\mathcal{R}$ := $\mathcal{R} \cup \{P\}$
33
34        elif result is a proof $P$ from
35            a skeleton-independent search
36            or a cycleQ:
37          {$P$ is a wall: $P$, $\neg P$ are SCC-closed}
38          if $m = 1$: {§C}
39            $d$ := singleCube($\mathcal{R}$, $\mathcal{C}$, $s_0$, $P$)
40            $\neg P$ := $\neg P \wedge \neg d$
41          $\mathcal{W}$ := $\mathcal{W} \cup \{P\}$
42          if heuristic($P$):
43            {$c_P$ is the choice variable for $P$}
44            if skelQ($\mathcal{R}$, $\mathcal{W}$, $\iota$, $K$) $\wedge\, c_P$ is unsat:
45              $C$ := $\neg P'$
46            elif skelQ($\mathcal{R}$, $\mathcal{W}$, $\iota$, $K$) $\wedge \neg c_P$ is unsat:
47              $C$ := $P$
48            else
49              $C$ := $P' \rightarrow P$
50            $\mathcal{C}$ := $\mathcal{C} \cup \{C\}$
51
52    return false {empty language}
```

Fig. 1. The fair algorithm: Does S have a computation?

Two forms of the skeleton query (skelQ) are used: the full query at lines 6, 44, and 46, based on the bijection ι between $\mathcal{B}$ and $\{1,\ldots,|\mathcal{B}|\}$ defined at line 4; and the skeleton-minimization version (Section III-B) at line 10, based on the map j defined at line 8. Notice that the latter version is only used to enforce a preference on skeletons and not, for example, to construct C at lines 44-49. In this pseudocode, all queries use the same K; however, it would be reasonable for the queries at lines 44 and 46 to use a different unrolling than K. In particular, since the full version has as many copies of T as $2K|\mathcal{B}|$, it may only be practical to use an unrolling of 0 or 1 for these queries, which are executed more frequently than the one at line 6.

Lines 13-16 correspond to finding a skeleton-independent proof (Section III-E); if none exist, then this choice is disabled. Lines 18-25 correspond to choosing a skeleton (Sections III-A and III-B) and executing the one stem (stemQ) and m cycle (cycleQ) queries (Section III-F).

Lines 27-50 act on the result of the search for a new proof. If all (safety) queries returned counterexample runs, then they can be formed into a "lasso" representing a computation of S (lines 27-28). Otherwise, if stemQ returned proof P, then P describes new reachability information (lines 30-32).

Otherwise, if either a skeleton-independent proof P is discovered (Section III-E) or a cycle query returned proof P, then P is a wall, and P and $\neg P$ are SCC-closed regions (lines 34-50). If the skeleton has just one state ($m = 0$) and $K = 0$, then it is necessary to augment $\neg P$ with additional information (Section III-C), and it might be useful to do so if $K > 0$ as well (lines 38-40). Line 40 takes liberties with logic: it says that $\neg P$ will henceforth be $\neg P \wedge \neg d$, so that $\neg P$ is no longer simply the negation of P. In other words, the list $\mathcal{W}$ of walls must actually be implemented as two lists, one to hold positive proofs and the other to hold possibly modified negative proofs. If P is determined heuristically to be interesting (line 42), then a C constraint is constructed and added to $\mathcal{C}$ (lines 42-50, Section III-A). Lines 44-45 correspond to the case in which no skeleton exists on the P side of the wall; lines 46-47 correspond to the case in which no skeleton exists on the $\neg P$ side of the wall; and lines 48-49 correspond to the typical case in which both sides have skeletons but the wall cannot be crossed.

If the skeleton-minimization version of the skeleton query at line 10 is unsatisfiable, then the full version is tested at line 6; if it is satisfiable, then a new map is constructed at line 8. If, however, the full skeleton query at line 6 is also unsatisfiable, then S does not have a computation (line 52).

H. Correctness

We prove the correctness of the fair algorithm. The first three lemmas formalize the assumption that the safety model checker is correct.

Lemma 1: A proof is returned for query (1) iff s_0 is unreachable from I, and such a proof excludes s_0 and is S-inductive relative to $\mathcal{R}$.

Hence, no subsequent skeleton contains s_0.

Lemma 2: A proof is returned for query (3) iff $s_{i\oplus_n 1}$ is unreachable from s_i, and such a proof separates s_i from $s_{i\oplus_n 1}$ and is S-inductive relative to $\mathcal{R}$ and $\mathcal{C}$, with the exception that it satisfies initiation with respect to s_i rather than I.

Hence, no subsequent skeleton contains both s_i and $s_{i\oplus_n 1}$.

Lemma 3: A proof is returned for query (4) iff s_0 is unreachable from its successors, and such a proof separates the successors of s_0 from s_0 and is S-inductive relative to $\mathcal{R}$ and $\mathcal{C}$, with the exception that it satisfies initiation with respect to the successors of s_0 rather than I.

Combined with either $K > 0$ for the technique of Section III-B or the technique of Section III-C to exclude s_0 from the $\neg P$ side of the wall, no subsequent skeleton contains s_0.

Besides progress criteria, these lemmas together imply that a skeleton can be completed into a reachable fair cycle if and only if all queries return counterexample runs.

Lemma 4: No transition excluded by a constraint $C \in \mathcal{C}$ is on a reachable fair cycle.

This lemma is straightforward once one realizes that each C is derived from (relatively) inductive information. A proof W from a cycle query observes that no path allowed by the current $\mathcal{C}$ that passes from a $\neg W$-state to a W-state can be part of a cycle, as it can never return to a $\neg W$-state. This observation is encoded as $W \leftrightarrow W'$. Additionally, if W-states ($\neg W$-states) cannot satisfy every fairness condition, then no path that has a W-state ($\neg W$-state) can be part of a fair cycle. This observation is encoded as $W \wedge W'$ ($\neg W \wedge \neg W'$). Hence, induction on the list $\mathcal{C}$ proves the lemma.

Another perspective on this lemma is that a cycle query proof W, by its inductiveness, describes regions W and $\neg W$ that are SCC-closed *with respect to S constrained by $\mathcal{C}$*. The resulting constraint C excludes only transitions leaving an SCC-closed region or all transitions of an SCC-closed region that does not intersect some fair condition; hence, no transition of a fair cycle is excluded.

By similar reasoning, one concludes that, in general, any fair cycle must be entirely contained in an arena defined by $\mathcal{W}$-constraints: for each $W \in \mathcal{W}$, the entire cycle must satisfy either W or $\neg W$. Hence we have the following lemma.

Lemma 5: If the skeleton query is unsatisfiable, then S does not have a reachable fair cycle.

Together these lemmas imply correctness of the algorithm.

Theorem 1: The algorithm fair always terminates, and it returns a reachable fair cycle iff the language of S is nonempty.

As suggested in Section III-A, the constraints $\mathcal{C}$ that are used during cycle queries are unnecessary for completeness, although crucial for the algorithm to be effective in practice. Lemma 4 states that these constraints do not destroy soundness. In contrast, all constraints in the skeleton query corresponding to the members of the sets $\mathcal{R}$ and $\mathcal{W}$ are necessary for completeness, as suggested by Lemmas 1-3, which state how the algorithm makes progress. Each new reachability assertion $R \in \mathcal{R}$ eliminates at least one state from being returned henceforth from a skeleton query; and each new wall $W \in \mathcal{W}$ eliminates at least one pair of states (Lemma 2) or one state (Lemma 3) from further consideration.

IV. Related Work

Several fair cycle detection algorithms have been developed for symbolic model checking. In this section we compare the main ones to fair, focusing on two features: the identification of SCC-closed sets and the ability to generalize from facts learned about the model.

SCC decomposition algorithms [11]–[13] recursively divide the states into SCC-closed sets. In that respect, they are the closest to fair. However, the walls that they derive are local to the arenas from which SCCs are extracted. Therefore, if the language of a model is empty, SCC decomposition must break up all reachable arenas until they become trivial or unfair. In contrast, fair produces wall that transect the entire state space; hence, it can prove language emptiness by considering a number of skeletons that is much smaller than the number of nontrivial SCCs.

SCC hull algorithms [14], [15] compute an SCC-closed set that contains all fair SCCs and that is empty if no fair SCC exists. In its simplest form, an SCC hull is defined by one wall. (See [15] for hulls defined by two walls.) One side of the wall is known to contain no fair SCC, and the algorithms move the wall until the SCCs abutting the wall on the other side are all fair. While the wall may be moved across very large numbers of SCCs in one step of the procedure, the restriction to a small, fixed set of walls prevents SCC hull algorithms from learning important facts about the structure of the SCC graph. In addition, SCC hull algorithms converge to a hull before declaring a language nonempty. In contrast, fair is often able to home in on a reachable fair SCC well before the entire state space has been examined. Every skeleton that is examined focuses the successive skeleton queries on where the fair SCCs may lie.

Among the first algorithms for BDD-based cycle detection is the one of [16] based on the computation of the transitive closure of the state graph by iterative squaring. The approach works well for counters, but unlike fair, it is often impractical because it computes too much information about the model.

In Bounded Model Checking (BMC) [9] cycle detection can be formulated as a SAT query such that a model of an appropriate formula describes a lasso-shaped path of prescribed length in the given finite-state system. Deciding that no lasso-shaped path exists regardless of length requires the computation of appropriate bounds (e.g., [17]). While this approach does not fix a skeleton in advance, failure to find a path of a given length does not directly translate into information about the SCC-closed sets of the model. By separating the choice of the skeleton from the attempt to flesh it out to a cycle, fair incrementally learns inductive lemmas.

The liveness-to-safety conversion of [3] is the most common approach to prove progress with interpolation-based model checking [4], [10]. While safety checking is more developed and arguably better understood than checking for progress properties, the transformation to safety has several drawbacks: first, the model's doubled number of state variables negatively affects some model checkers; second, the nature of the problem—cycle detection—is not obvious to the model checker from the encoding; third, the approach is inherently non-incremental, because it asks the safety model checker for a single, monolithic proof that there is no fair cycle.

In the D'n'C approach [18], SCC decomposition is applied to a sequence of increasingly refined abstractions of a system. If an effective way to choose the abstract models is given, this approach may be profitably combined with fair to initially provide it with simple lemmas about the abstractions. Both methods can leverage the weakness of fairness conditions; fair, however, can sometimes discover weakness even on large structures—even, that is, when weakness is not inherited from the acceptance condition of a small Büchi automaton.

V. Experimental Evaluation

An implementation of fair was evaluated against other cycle detection methods on a set of models. Even though fair is highly parallelizable, the implementation uses only one thread of execution but employs a time sharing scheme, as described in Section III-F.

The implementation of skeleton queries differs from the description of Section III-B: for $K = 0$, one forward and no backward unrolling is used; for $K = 1$, two forward and one backward unrollings are used; and so on. Therefore, it only adds a clause as in Section III-C if it provides additional information.

The skeleton-minimization heuristic of Section III-B is implemented as a search: map construction is guided by intermediate partial skeleton queries based on partial maps. If a partial map corresponds to an unsatisfiable query, the last assignment of an index to a fairness condition is incremented, potentially extending the range of the partial map by one. Of course, if the assignment is already onto $\{1, \ldots, |\mathcal{B}|\}$, then the standard skeleton query is also unsatisfiable, and the proof is complete. Once a map is constructed, it is used until the corresponding skeleton query becomes unsatisfiable, at which point a new map is constructed. A separate full skeleton query is used throughout execution, as described in Section III-G.

The implementation also checks if each proof returned by a cycle query is actually inductive with respect to the system, and if so, the proof is upgraded to a reachability proof. While the benchmarks did not reveal if this check is worthwhile, it is inexpensive. Finally, only IC3 is used to answer safety queries, and its proofs are refined as described in Section III-D.

Unlike the case of safety properties, there are no widely accepted benchmark sets for progress properties. Moreover, models of practical import are difficult to come by. The evaluation therefore relies on models that have been identified in the literature as challenging for certain approaches or that present features that one may find combined in real-life problem instances. The abq, cnt, and jc models were written for this evaluation; the remaining ones were adapted from [19].

Table I reports the results of the experiments, which were run on machines with one 2.67 GHz Intel Core i5 CPU and 8 GB of memory each. CPU times are in seconds. The timeout was set at 7200 s. For each model, the table shows whether the language is empty, the number of latches in the cone of influence of the fairness conditions, the number of 2-input AND gates after combinational simplification, and the number of fairness conditions (with the number of weak conditions in parentheses). Next, the results for fair are shown: in the latter three columns, for $0 \leq K \leq 2$ with the skeleton-minimization heuristic enabled, the CPU time and the number of skeletons examined are reported. If fair timed out (indicated by a dash) the number of skeletons examined up to that point is given. The first column for fair shows similar results for $K = 0$ with the skeleton-minimization heuristic disabled.

The remaining columns show results for other language emptiness algorithms. GSH, LS, and DnC are the SCC hull method of [15], the SCC decomposition method of [12], and the D'n'C algorithm of [18] as implemented in the lang_empty command of VIS 2.3 [20] (run with dynamic variable ordering enabled and default settings except that D'n'C is run without preliminary reachability analysis). These three methods were chosen for inclusion in the table because they represent well the gamut of BDD-based algorithms and because GSH and D'n'C without reachability performed better than the others that were tried.

Finally, the group of columns under LTS refers to the liveness-to-safety approach of [3], with reachability checked with interpolation as implemented in ABC [4] (ITP), with IC3, and with ABC. For ITP, the parameters controlling ABC were set to disable its IC3 implementation and to reduce the chance of inconclusive runs. A question mark in the ITP column signals that ABC nevertheless reported the problem as "undecided" before its time was up. For ABC, the parameter controlling its use of its IC3 implementation was set to allow

TABLE I
EXPERIMENTS

					fair				BDD-based			LTS		
model	empty	latches	gates	$\lvert\mathcal{B}\rvert$	$K = 0^*$	$K = 0$	$K = 1$	$K = 2$	GSH	LS	DnC	ITP	IC3	ABC
abq2mf	yes	35	383	4(1)	1/24	1/12	1/8	1/5	1	1	1	–	2	8
abq4mf	yes	67	745	6(1)	3/37	3/39	2/8	3/9	3	–	7	–	11	40
abq8mf	yes	131	1469	10(1)	23/182	168/67	16/14	21/14	2794	–	–	–	373	157
abq2f	yes	61	747	4(1)	3/30	3/55	2/9	4/3	4	10	1	–	10	20
abq4f	yes	119	1471	6(1)	423/221	31/106	13/28	34/46	2890	–	213	–	388	–
abq8f	yes	235	2923	10(1)	–/75	–/116	5730/84	4384/65	–	–	–	–	6330	–
cnt12	yes	12	68	1(1)	1/0	1/0	1/0	1/0	1	1	0	1	1761	1
cnt32	yes	32	188	1(1)	1/0	1/0	1/0	1/0	–	–	–	?	–	–
cnt128	yes	128	764	1(1)	1/0	1/0	1/0	1/0	–	–	–	?	–	–
jc12	yes	13	231	1(1)	1/0	1/0	1/0	1/0	1	1	0	9	93	9
jc32	yes	33	631	1(1)	1/0	1/0	1/0	1/0	–	–	–	16	–	–
jc128	yes	129	2551	1(1)	2/0	2/0	2/0	3/0	–	–	–	805	–	–
jc128f	no	129	2170	1(1)	2/1	2/1	2/1	2/1	2	2	2	1	1	1
om1	yes	29	810	16(16)	–/99	–/202	–/244	–/274	272	–	356	–	–	–
om2	yes	29	806	16(16)	42/2082	39/2077	42/2083	45/2071	192	–	8	–	236	–
om3	yes	29	803	16(16)	1/0	1/0	2/0	5/0	35	–	25	–	105	–
nim1	yes	27	769	2(2)	1/29	1/32	1/0	1/0	1	174	1	–	20	117
nim2	yes	29	788	2(2)	1309/28	1264/28	1157/18	1457/18	2	120	1	–	1192	177
nim3	no	29	788	2(2)	1/32	1/28	1/3	1/3	1	309	1	1	1	1
gbak	yes	37	677	10(1)	25/182	12/172	74/184	26/125	3	7	14	–	97	90
tarb16	yes	79	1109	17(1)	18/166	15/101	17/72	70/79	–	–	–	60	58	31
tarb32	yes	159	2269	33(1)	146/582	75/204	214/146	956/147	–	–	–	?	–	209
sarb16	yes	50	141	1(1)	1/0	1/0	1/0	1/0	1	1	3	?	5	7
sarb32	yes	98	269	1(1)	1/0	1/0	1/0	1/0	3	1	–	?	157	79
tf1	yes	30	452	2(1)	1949/5174	393/2222	285/1278	288/1213	8	–	2	–	–	281
tf2	no	30	384	2(1)	1/9	1/5	1/2	1/2	2	60	1	1	1	1
tq1	yes	55	756	3(1)	2267/3072	3143/4208	2690/2775	5434/3172	1645	–	3	–	–	737
tq2	no	60	771	4(2)	5/27	4/28	4/22	5/26	3056	–	5	2	27	2
fq1	yes	105	1365	5(1)	–/2920	–/2596	–/2485	–/1771	–	–	336	–	–	–
fq2	no	120	1546	8(4)	21/41	15/39	25/55	29/44	–	–	–	–	374	30

it to run through the two-hour time limit.

The abq models are interconnected queues with bounded sources. The cnt models are counters and the jc models are the "forward jumping counters" of [3]. The om models are used in [15] to prove lower bounds on SCC hull algorithms. The nim models are NIM players. The gbak model is a finite-state version of the bakery protocol. The tarb models are tree arbiters, while the sarb models are McMillan synchronous arbiters. The tf, tq, and fq models are versions of the two-queue example in [21].

The cnt models illustrate fair's ability to find linear-size proofs for counters as discussed in Section III-E. This ability accounts for the good performance of fair on models like the om and nim (NIM player) sets—in which the original state graph has many SCCs—or like the jc and tarb (tree arbiter) sets, in which the composition with a Büchi automaton breaks the single SCC of the model into a myriad of SCCs. While computing the transitive closure would be effective for counters, it would not work on more complex examples.

The om set contains three models that differ only in the transitions out of unreachable states. For om3, fair quickly produces an inductive proof that there are no fair SCCs; for the other two models, however, it has to prove, at a much higher cost, that such fair SCCs are unreachable. Combining fair with a global reachability engine, perhaps based on BDDs, would benefit the analysis for om1 and om2, but was outside the scope of this evaluation. Yet not relying on full reachability analysis is partly responsible for fair's speed in detecting nonemptiness for tq2 and fq2.

For all four configurations, fair decided either 27 or 28 of the 30 language emptiness problems and was the only model checker to solve two of the problems. Behind it, each of GSH, DnC, and LTS/IC3 solved 21 problems, and LTS/ABC solved 20 problems. Together the BDD methods solved 22 problems, and the LTS methods solved 26 problems. On 11 models, one of the fair configurations, typically $K = 1$, was decidedly faster than the other methods; on 8 models, one of the other six methods was decidedly faster. Overall, fair was the clear winner on this set of models.

It is worth noting that the two models that fair failed to prove—om1 and fq1—were solved by BDD methods but not by LTS methods. Furthermore, fair generally dominated the LTS methods, with the exception of nim2 and tq1, both of which were, in any case, trivial for at least one BDD method. In short, fair seems to complement BDD methods and to dominate LTS methods.

As expected, LS suffered on models with many SCCs, while LTS/ITP had rather unpredictable performance. For many models the number of skeletons examined by fair decreased with increasing K, with the largest jump usually occurring between $K = 0$ and $K = 1$; however, on tarb16 and tarb32, which have many fairness conditions and thus require large skeleton queries, fair suffered as K increased.

The skeleton-minimization heuristic of Section III-B is effective at finding small skeletons. For $K = 0$, six of the models on which the analysis finished required examining skeletons that have more than one state: nim1 (≤ 2), gbak (always 3, as there are three disjoint fairness conditions), tarb32 (≤ 2), tq1 (≤ 2), tq2 (≤ 2), fq2 (≤ 4). Furthermore, it typically resulted in fewer skeletons, as hypothesized; tarb32 and tf1 are extreme cases.

These illustrative benchmarks indicate the potential of the fair algorithm. However, only practical experience with a suite of industrial benchmarks will reveal the best use of skeleton-minimization, a method for choosing K dynamically, and a heuristic for choosing when to enrich the $\mathcal{C}$ constraint set.

VI. Conclusion

We have presented a new incremental algorithm for model checking progress properties that selects skeletons for fair cycles and, if it fails to flesh them out, learns inductive lemmas that divide the states into SCC-closed sets. An initial implementation shows promise, especially when one considers that one of the strengths of the proposed approach—that of being highly parallelizable—was not brought into play. Another important aspect that awaits exploration is the integration of the new approach into a multi-engine framework, which has been shown to be key to robust performance in the case of safety properties.

Acknowledgments. The first author thanks Barbara Jobstmann for a collaboration that inspired this work while the two were post-docs in Tom Henzinger's group at EPFL. Thanks also to the reviewers for their specific questions and suggestions, which aided us in improving the presentation. This material is based upon work supported in part by the National Science Foundation under grant No. 0952617 and by the Semiconductor Research Corporation under contract GRC 1859. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, Austin, TX, 2011, pp. 70–87, lNCS 6538.

[2] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[3] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," *Software Tools for Technology Transfer*, vol. 5, no. 2–3, pp. 185–204, Mar. 2004.

[4] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Twentysecond Conference on Computer Aided Verification (CAV"10)*. Edinburgh, UK: Springer, 2010, pp. 24–40, lNCS 6174.

[5] D. E. Muller, A. Saoudi, and P. E. Schupp, "Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time," in *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, Edinburgh, UK, Jul. 1988, pp. 422–427.

[6] R. Bloem, K. Ravi, and F. Somenzi, "Efficient decision procedures for model checking of linear time logic properties," in *Eleventh Conference on Computer Aided Verification (CAV'99)*, N. Halbwachs and D. Peled, Eds. Berlin: Springer-Verlag, 1999, pp. 222–235, lNCS 1633.

[7] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer Aided Design (FMCAD'07)*, Austin, TX, 2007, pp. 173–180.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[9] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, Amsterdam, The Netherlands, Mar. 1999, pp. 193–207, lNCS 1579.

[10] K. L. McMillan, "Interpolation and SAT-based model checking," in *Fifteenth Conference on Computer Aided Verification (CAV'03)*, W. A. Hunt, Jr. and F. Somenzi, Eds. Berlin: Springer-Verlag, Jul. 2003, pp. 1–13, lNCS 2725.

[11] A. Xie and P. A. Beerel, "Implicit enumeration of strongly connected components and an application to formal verification," *IEEE Transactions on Computer-Aided Design*, vol. 19, no. 10, pp. 1225–1230, Oct. 2000.

[12] R. Bloem, H. N. Gabow, and F. Somenzi, "An algorithm for strongly connected component analysis in $n \log n$ symbolic steps," *Formal Methods in System Design*, vol. 28, no. 1, pp. 37–56, Jan. 2006.

[13] R. Gentilini, C. Piazza, and A. Policriti, "Symbolic graphs: Linear solutions to connectivity related problems," *Algorithmica*, vol. 50, no. 1, pp. 120–158, 2008.

[14] E. A. Emerson and C.-L. Lei, "Efficient model checking in fragments of the propositional mu-calculus," in *Proceedings of the First Annual Symposium of Logic in Computer Science*, Jun. 1986, pp. 267–278.

[15] F. Somenzi, K. Ravi, and R. Bloem, "Analysis of symbolic SCC hull algorithms," in *Formal Methods in Computer Aided Design*, M. D. Aagaard and J. W. O'Leary, Eds. Springer-Verlag, Nov. 2002, pp. 88–105, lNCS 2517.

[16] H. J. Touati, R. K. Brayton, and R. P. Kurshan, "Testing language containment for ω-automata using BDD's," *Information and Computation*, vol. 118, no. 1, pp. 101–109, Apr. 1995.

[17] M. Awedh and F. Somenzi, "Termination criteria for bounded model checking: Extensions and comparison," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 1, pp. 51–66, 2006, presented at the Third International Workshop on Bounded Model Checking (BMC'05).

[18] C. Wang, R. Bloem, G. D. Hachtel, K. Ravi, and F. Somenzi, "Divide and compose: SCC refinement for language emptiness," in *International Conference on Concurrency Theory (CONCUR01)*. Berlin: Springer-Verlag, Aug. 2001, pp. 456–471, lNCS 2154.

[19] "VIS verification benchmarks. http://vlsi.colorado.edu/∼vis," University of Colorado at Boulder.

[20] "URL: http://vlsi.colorado.edu/∼vis."

[21] H. Jin, K. Ravi, and F. Somenzi, "Fate and free will in error traces," *Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 102–116, Aug. 2004.