Certifying Model Checkers

Kedar S. Namjoshi

Bell Laboratories, Lucent Technologies
kedar@research.bell-labs.com
http://www.cs.bell-labs.com/who/kedar

Abstract. Model Checking is an algorithmic technique to determine whether a temporal property holds of a program. For linear time properties, a model checker produces a counterexample computation if the check fails. This computation acts as a "certificate" of failure, as it can be checked easily and independently of the model checker by simulating it on the program. On the other hand, no such certificate is produced if the check succeeds. In this paper, we show how this asymmetry can be eliminated with a *certifying* model checker. The key idea is that, with some extra bookkeeping, a model checker can produce a *deductive proof* on either success or failure. This proof acts as a certificate of the result, as it can be checked mechanically by simple, non-fixpoint methods that are independent of the model checker. We develop a deductive proof system for verifying branching time properties expressed in the mu-calculus, and show how to generate a proof in this system from a model checking run. Proofs for linear time properties form a special case. A model checker that generates proofs can be used for many interesting applications, such as better ways of exploring errors in a program, and a tight integration of model checking with automated theorem proving.

1 Introduction

Model Checking [CE81,QS82] is an algorithmic technique to determine whether a temporal property holds of a program. Perhaps the most useful property of the model checking algorithm is that it can generate a counterexample computation if a linear time property fails to hold of the program. This computation acts as a "certificate" of failure, as it can be checked easily and efficiently by a method independent of model checking – i.e., by simulating the program to determine whether it can generate the computation. On the other hand, if it is determined that a property holds, model checkers produce only the answer "yes"! This does not inspire the same confidence as a counterexample; one is forced to assume that the model checker implementation is correct. It is desirable, therefore, to provide a mechanism that generates certificates for either outcome of the model checking process. These certificates should be easily checkable by methods that are independent of model checking.

In this paper, we show how such a mechanism, which we call a *certifying model checker*, can be constructed. The key idea is that, with some extra bookkeeping, a model checker can produce a *deductive proof* on either success or failure. The proof acts as a certificate of the result, since it can be checked independently using simple, non-fixpoint methods. A certifying model checker thus provides a bridge from the "model-theoretic" to the "proof-theoretic" approach to verification [Eme90].

We develop a deductive proof system for verifying mu-calculus properties of programs, and show it to be sound and relatively complete. We then show how to construct a deductive proof from a model checking run. This is done by by storing and analyzing sets of states that are generated by the fixpoint computations performed during model checking. The proof system and the proof generation process draw upon results in [EJ91] and [EJS93], which relate model checking for the mu-calculus to winning parity games. A prototype implementation of a proof generator and proof checker for linear time properties has been developed for the COSPAN [HHK96] symbolic model checker.

The ability to generate proofs which justify the outcome of model checking makes possible several interesting applications. For instance,

- A certifying model checker produces a proof of property f on success, and a proof of $\neg f$ on failure. The proof of $\neg f$ is a compact representation of *all* possible counterexample computations. As is shown later, it can be exponentially more succinct than a single computation. Particular counterexample computations can be "unfolded" out of the proof by an interactive process which provides a better understanding of the flaws in the program than is possible with a single computation.
- Producing a deductive proof makes it possible to tightly integrate a certifying model checker into an automated theorem prover. For instance, the theorem prover can handle meta-reasoning necessary for applying compositional or abstraction methods, while checking subgoals with a certifying model checker. The proofs produced by the model checker can be composed with the other proofs to form a single, checkable, proof script.

The paper is organized as follows. Section 2 contains background information on model checking and parity games. Section 3 develops the deductive proof system for verifying mu-calculus properties, and Section 4 shows how such proofs can be generated by slightly modifying a mu-calculus model checker. Applications for certifying model checkers are discussed in detail in Section 5. Section 6 concludes the paper with a discussion of related work.

2 Preliminaries

In this section, we define the mu-calculus and alternating tree automata, and show how mu-calculus model checking can be reduced to determining winning strategies in parity games.

2.1 The mu-calculus

The mu-calculus [Koz82] is a branching time temporal logic that subsumes [EL86] commonly used logics such as LTL, ω-automata, CTL, and CTL*. The logic is parameterized with respect to two sets: Σ (state labels) and Γ (action labels). There is also a set of variable symbols, V. Formulas of the logic are defined using the following grammar, where l is in Σ, a is in Γ, Z is in V, and μ is the least fixpoint operator.

$\Phi ::= l \mid Z \mid \langle a \rangle \Phi \mid \neg \Phi \mid \Phi \wedge \Phi \mid (\mu Z : \Phi)$

To simplify notation, we assume that Σ and Γ are fixed in the rest of the paper. A formula must have each variable under the scope of an even number of negation symbols. A formula is *closed* iff every variable in it is under the scope of a μ operator. Formulas are evaluated over labeled transition systems (LTS's) [Kel76]. An LTS is a tuple (S, s_0, R, L), where S is a non-empty set of *states*, $s_0 \in S$ is the *initial state*, $R \subseteq S \times \Gamma \times S$ is the *transition relation*, and $L : S \rightarrow \Sigma$ is a *labeling function* on states. We assume that R is *total*; i.e., for any s and a, there exists t such that $(s, a, t) \in R$. The evaluation of a formula f, represented as $\|f\|_c$, is a subset of S, and is defined relative to a *context* c mapping variables to subsets of S. The evaluation rule is given below.

- $\|l\|_c = \{s | s \in S \wedge L(s) = l\}$, $\|Z\|_c = c(Z)$,
- $\|\langle a \rangle \Phi\|_c = \{s | (\exists t : R(s, a, t) \wedge t \in \|\Phi\|_c)\}$,
- $\|\neg \Phi\|_c = S \backslash \|\Phi\|_c$, $\|\Phi_1 \wedge \Phi_2\|_c = \|\Phi_1\|_c \cap \|\Phi_2\|_c$,
- $\|(\mu Z : \Phi)\|_c = \bigcap\{T : T \subseteq S \wedge \|\Phi\|_{c[Z \leftarrow T]} \subseteq T\}$, where $c[Z \leftarrow T]$ is the context c' where, for any X, $c'(X)$ is T if $X = Z$, and $c(X)$ otherwise.

A state s in the LTS *satisfies* a closed mu-calculus formula f iff $s \in \|f\|_\perp$, where $\perp$ maps every variable to the empty set. The LTS satisfies f iff s_0 satisfies f. Mu-calculus formulas can be converted to positive normal form by introducing the operators $\Phi_1 \vee \Phi_2 = \neg(\neg(\Phi_1) \wedge \neg(\Phi_2))$, $[a]\Phi = \neg\langle a \rangle(\neg \Phi)$ and $(\nu Z : \Phi) = \neg(\mu Z : \neg \Phi(\neg Z))$, and using de Morgan rules to push negations inwards. The result is a formula where negations are applied only to elements of Σ.

Mu-calculus Signatures: Consider a closed mu-calculus formula f in positive normal form, where the μ-variables are numbered $Y_1, \ldots, Y_n$ in such a way that if (μY_i) occurs in the scope of (μY_j) then $j < i$. Streett and Emerson [SE84] show that, with every state s of an LTS M that satisfies f, one can associate a lexicographically minimum n-vector of ordinals called its *signature*, denoted by $sig(s, f)$. Informally, $sig(s, f)$ records the minimum number of unfoldings of least fixpoint operators that are necessary to show that s satisfies f. For example, for the CTL property $\mathsf{EF}(p) = (\mu Y_1 : p \vee \langle \tau \rangle Y_1)$, $sig(s, \mathsf{EF}(p))$ is the length of the shortest τ-path from s to a state satisfying p.

Formally, for an n-vector of ordinals v, let f^v be a formula with the semantics defined below. Then $sig(s, f)$ is the smallest n-vector v such that $s \in \|f^v\|_\perp$. First, define the new operator μ^k, for an ordinal k, with the semantics $\|(\mu^k Y : \Phi)\|_c = Y^k$, where $Y^0 = \emptyset$, $Y^{i+1} = \|\Phi\|_{c[Y \leftarrow Y^i]}$, and for a limit ordinal λ, $Y^\lambda = (\cup k : k < \lambda : Y^k)$.

- $\|l^v\|_c = \|l\|_c$, $\|(\neg l)^v\|_c = \|\neg l\|_c$, $\|Z^v\|_c = \|Z\|_c$,
- $\|(\langle a \rangle \Phi)^v\|_c = \|\langle a \rangle(\Phi^v)\|_c$, $\|([a]\Phi)^v\|_c = \|[a](\Phi^v)\|_c$,
- $\|(\Phi_1 \wedge \Phi_2)^v\|_c = \|\Phi_1^v \wedge \Phi_2^v\|_c$, $\|(\Phi_1 \vee \Phi_2)^v\|_c = \|\Phi_1^v \vee \Phi_2^v\|_c$
- $\|(\mu Y_j : \Phi)^v\|_c = \|\Phi^v\|_{c'}$, where $c' = c[Y_j \leftarrow \|(\mu^{v[j]} Y_j : \Phi)\|_c]$
- $\|(\nu Z : \Phi)^v\|_c = \|\Phi^v\|_{c'}$, where $c' = c[Z \leftarrow \|(\nu Z : \Phi)\|_c]$.

2.2 Alternating Automata and Parity Games

An alternating automaton is another way of specifying branching time temporal properties. For sets Σ and Γ of state and transition labels respectively, an alternating automaton is specified by a tuple (Q, q_0, δ, F), where Q is a non-empty set of states, $q_0 \in Q$ is the initial state, and δ is a transition function mapping a pair from $Q \times \Sigma$ to a positive boolean expression formed using the operators $\wedge, \vee$ applied to elements of the form $\mathit{true}, \mathit{false}, q, \langle a \rangle q$ and $[a]q$, where $a \in \Sigma$, and $q \in Q$. F is a *parity* acceptance condition, which is a non-empty list $(F_0, F_1, \ldots, F_n)$ of subsets of Q. An infinite sequence over Q satisfies F iff the smallest index i for which a state in F_i occurs infinitely often on the sequence is even. For simplicity, we assume that the transition relation of the automaton is in a normal form, where F is a partition of Q, and $\delta(q, l)$ has one of the following forms: $q_1 \wedge q_2, q_1 \vee q_2, \langle a \rangle q_1, [a]q_1, \mathit{true}, \mathit{false}$. Converting an arbitrary automaton to an equivalent automaton in normal form can be done with a linear blowup in the size.

A *tree* is a prefix-closed subset of $\mathbb{N}^*$, where λ, the empty sequence, is called the *root* of the tree. A labeled tree t is a tree together with two functions $N_t : t \to \Sigma$ and $E_t : \mathit{edge}(t) \to \Gamma$, where $\mathit{edge}(t) = \{(x, x.i) | x \in t \wedge x.i \in t\}$. We require the transition relation of such a tree to be total.

The acceptance of a labeled tree t by the automaton is defined in terms of a two-player infinite game. A *configuration* of the game is a pair (x, q), where x is a node of the tree and q is an automaton state. If $\delta(q, N_t(x))$ is true, player I wins, while player II wins if it is false. For the other cases, player I chooses one of q_1, q_2 if it is $q_1 \vee q_2$, and chooses an a-successor to x if it is $\langle a \rangle q_1$. Player II makes similar choices at the $\wedge$ and $[a]$ operators. The result is a new configuration (x', q'). A *play* of the game is a maximal sequence of configurations generated in this manner. A play is winning for player I iff either it is finite and ends in a configuration that is a win for I, or it is infinite and satisfies the automaton acceptance condition. The play is winning for player II otherwise. A *strategy* for player I (II) is a partial function that maps every finite sequence of configurations and intermediate choices to a choice at each player I (II) position. A *winning* strategy for player I is a strategy function where every play following that strategy is winning for I, regardless of the strategy for II. The automaton *accepts* the tree t iff player I has a winning strategy for the game starting at (λ, q_0). An LTS M *satisfies* the automaton iff the automaton accepts the computation tree of M.

Theorem 0. [EJ91,JW95] For any closed mu-calculus formula f, there is a linear-size alternating automaton A_f such for any LTS M, M satisfies f iff M satisfies A_f. The automaton is derived from the parse graph of the formula.

A strategy s is *history-free* iff the outcome of the function depends only on the last element of the argument sequence. By results in [EJ91], parity games are determined (one of the players has a winning strategy), and the winner has a history-free winning strategy. From these facts, winning in the parity game generated by an LTS $M = (S, s_0, R, L)$ and an automaton $A = (Q, q_0, \delta, F)$

can be cast as model checking on a product LTS, $M \times A$, of configurations [EJS93]. The LTS $M \times A = (S', s'_0, R', L')$ is defined over state labeling $\Sigma' = \{I, II, win_I, win_{II}\} \times \{f_0, \ldots, f_n\}$ and edge labeling $\Gamma' = \{\tau\}$, and has $S' = S \times Q$ and $s'_0 = (s_0, q_0)$. The first component of $L'(s,q)$ is I if $\delta(q, L(s))$ has the form $q_1 \vee q_2$ or $\langle a \rangle$, II if it has the form $q_1 \wedge q_2$ or $[a]q_1$, win_I if it has the form *true*, and win_{II} if it has the form *false*. The second component is f_i iff $q \in F_i$. R' is defined as follows. For a state (s,q), if $\delta(q, L(s))$ is *true* or *false*, then (s,q) has no successors; if $\delta(q, L(s))$ is $q_1 \vee q_2$ or $q_1 \wedge q_2$, then (s,q) has two successors (s, q_1) and (s, q_2); if $\delta(a, L(s))$ is $\langle a \rangle q_1$ or $[a]q_1$, then (s,q) has a successor (t, q_1) for every t such that $R(s,a,t)$ holds, and no other successors.

Let $\mathcal{W}_I = (\sigma_0 Z_0 \ldots \sigma_n Z_n : \Phi_I(Z_0, \ldots, Z_n))$, where σ_i is ν if i is even and μ otherwise, and $\Phi_I(Z_0, \ldots, Z_n) = win_I \vee (I \wedge (\wedge i : f_i \Rightarrow \langle \tau \rangle Z_i)) \vee (II \wedge (\wedge i : f_i \Rightarrow [\tau] Z_i))$. The formula $\mathcal{W}_I$ describes the set of configurations from which player I has a winning strategy. Similarly, player II has a winning strategy from the the complementary set $\mathcal{W}_{II}$, where $\mathcal{W}_{II} = (\delta_0 Z_0 \ldots \delta_n Z_n : \Phi_{II}(Z_0, \ldots, Z_n))$, where δ_i is μ if i is even, and ν otherwise, and $\Phi_{II}(Z_0, \ldots, Z_n) = win_{II} \vee (I \wedge (\wedge i : f_i \Rightarrow [\tau] Z_i)) \vee (II \wedge (\wedge i : f_i \Rightarrow \langle \tau \rangle Z_i))$.

Theorem 1. (cf. [EJS93]) For an LTS M and a normal form automaton A of the form above, M satisfies A iff $M \times A, (s_0, q_0) \models \mathcal{W}_I$.

3 The proof system

Deductive proof systems for verifying sequential programs rely on the two key concepts of *invariance* (e.g., loop invariants) and *progress* (e.g., rank functions, variant functions) [Flo67,Hoa69]. These concepts reappear in deductive verification systems for linear temporal logic [MP83,MP87,CM88], and also form the basis for the proof system that is presented below.

Suppose that $M = (S, s_0, R, L)$ is an LTS, and $A = (Q, q_0, \delta, F)$ is a normal form automaton, where $F = (F_0, F_1, \ldots, F_{2n})$. To show that M satisfies A, one exhibits (i) for each automaton state q, a predicate (the invariant) ϕ_q over S, expressed in some *assertion language*, (ii) non-empty, well founded sets $W_1, \ldots, W_n$ with associated partial orders $\preceq_1, \ldots, \preceq_n$, and (iii) for each automaton state q, a partial rank function $\rho_q : S \to (W, \preceq)$, where $W = W_1 \times \ldots \times W_n$ and $\preceq$ is the lexicographic order defined on W using the $\{\preceq_i\}$ orders.

We extend the $\preceq$ order to apply to elements a, b in $W_1 \times \ldots \times W_k$, for some $k < n$ by $a \preceq b$ iff $(a_1, \ldots, a_k, 0, 0, \ldots, 0) \preceq (b_1, \ldots, b_k, 0, 0, \ldots, 0)$, where we assume, without loss of generality, that 0 is an element common to all the W_i's. For an automaton state q, define the relation $\lhd_q$ over $W \times W$ as follows. For any a, b, $a \lhd_q b$ holds iff for the (unique, since F is a partition) index k such that $q \in F_k$, either $k = 0$, or $k > 0, k = 2i$ and $(a_1, \ldots, a_i) \preceq (b_1, \ldots, b_i)$, or $k = 2i - 1$ and $(a_1, \ldots, a_i) \prec (b_1, \ldots, b_i)$. We use the label l to denote the predicate $l(s) \equiv (L(s) = l)$, and the notation $[f]$ to mean that the formula f is valid. Note that, in the following, $\langle a \rangle$ and $[a]$ are operators interpreted on M. The invariants and rank function must satisfy the following three local conditions. In these conditions, the variable k has type W.

- ***Consistency:*** For each $q \in Q$, $[\phi_q \Rightarrow (\exists k : (\rho_q = k))]$ (ρ_q is defined for every state in ϕ_q)
- ***Initiality:*** $\phi_{q_0}(s_0)$ (the initial state satisfies its invariant)
- ***Invariance and Progress:*** For each $q \in Q$, and $l \in \Sigma$, depending on the form of $\delta(q, l)$, check the following.
  - *true*: there is nothing to check.
  - *false*: $[\phi_q \Rightarrow \neg l]$ holds,
  - $q_1 \wedge q_2$: $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow (\phi_{q_1} \wedge (\rho_{q_1} \lhd_q k)) \wedge (\phi_{q_2} \wedge (\rho_{q_2} \lhd_q k))]$
  - $q_1 \vee q_2$: $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow (\phi_{q_1} \wedge (\rho_{q_1} \lhd_q k)) \vee (\phi_{q_2} \wedge (\rho_{q_2} \lhd_q k))]$
  - $\langle a \rangle q_1$: $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow \langle a \rangle(\phi_{q_1} \wedge (\rho_{q_1} \lhd_q k))]$
  - $[a]q_1$: $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow [a](\phi_{q_1} \wedge (\rho_{q_1} \lhd_q k))]$

Theorem 2. (Soundness) The proof system is sound.

Proof. Given a proof in the format above, we have to show that M satisfies A. We do so by exhibiting a winning strategy for player I in the parity game. For a configuration (s, q), let $\rho_q(s)$ be its associated rank. Inductively, assume that at any configuration (s, q) on a play, $\phi_q(s)$ is true. This holds at the start of the game by the Initiality requirement. Suppose that $L(s) = l$. Based on the form of $\delta(q, l)$, we have the following cases:

- *true*: the play terminates with a win for player I,
- *false*: this case cannot arise, as the inductive invariant contradicts the proof assertion $[\phi_q \Rightarrow \neg l]$.
- $q_1 \wedge q_2, [a]q_1$: Player II plays at this point, with the new configuration satisfying the inductive hypothesis by the proof.
- $q_1 \vee q_2$: Player I chooses the q_i for which the $\vee$ proof assertion holds. The new configuration (s, q_i) thus satisfies the inductive hypothesis.
- $\langle a \rangle q_1$: Player I chooses the a-successor t of s which is a witness for the $\langle a \rangle$ formula. Hence, $\phi_{q_1}(t)$ holds.

Thus, a finite play terminates with $\delta(q, l) = \mathit{true}$, which is a win for player I. In an infinite play, by the definition of $\lhd_q$, whenever the play goes through a configuration (s, q) with q in a odd-indexed set F_{2i-1}, the rank decreases strictly in the positions $1..i$, and the only way it can increase in these components is if the play later goes through a configuration (s', q') with q' in an even indexed set of smaller index. So, if an odd indexed set occurs infinitely often, some even indexed set with smaller index must also occur infinitely often, which implies that the smallest index that occurs infinitely often must be even. Thus, the defined strategy is winning for player I, so M satisfies A. □

Theorem 3. (Completeness) The proof system is relatively complete.

Proof. We show completeness relative to the expressibility of the winning sets, as is done for Hoare-style proof systems for sequential programs [Coo78]. Assume that M satisfies A. By Theorem 1, $M \times A, (s_0, q_0) \models \mathcal{W}_I$. The history-free winning strategy for player I corresponds to a sub-structure N of $M \times A$, which has a single outgoing edge at each player I state.

For each automaton state q, let $\phi_q(s) \equiv (M \times A, (s,q) \models \mathcal{W}_I)$. The rank function is constructed from the mu-calculus signatures of states satisfying the formula $\mathcal{W}_I$. For each automaton state q, let the function ρ_q have domain ϕ_q. For every state (s,q) satisfying $\mathcal{W}_I$, let $\rho_q(s)$ be the n-vector that is the signature of $\mathcal{W}_I$ at (s,q).

We now show that all the conditions of the proof rule are satisfied for these choices. Consistency holds by the definition of the ρ_q functions. Initiality holds by the definition of ϕ_q, since (s_0, q_0) satisfies $\mathcal{W}_I$. From the definition of signatures and the shape of the formula defining $\mathcal{W}_I$, it it not difficult to show that at each transition from a state in N, the signature for $\mathcal{W}_I$ decreases strictly in the first i components if the state is in F_{2i-1}, and is non-increasing in the first i components if the state is in F_{2i}. This corresponds directly to the progress conditions in the proof rule. For each state (s,q) in N, $\phi_q(s)$ is true, so the invariance conditions also hold. If $\delta(q,l)$ is *false*, then for any state s with $L(s) = l$, (s,q) represents a win for player II, so that $s \notin \mathcal{W}_I$, and $\phi_q \wedge l$ is unsatisfiable, as desired. $\square$

Proofs for linear time properties: Manna and Pnueli [MP87] show that every ω-regular linear time property can be represented by a $\forall$-automaton, which accepts an ω-string iff *all* runs of the automaton on the string satisfy a co-Büchi acceptance condition. Model checking the linear time property h is equivalent to checking the branching time property $\mathsf{A}(h)$. By the $\forall$ nature of acceptance, the A quantifier can, informally, be distributed through h, resulting in a tree automaton where δ is defined using only $[a]$ and $\wedge$ operators. Specializing our proof system to such automata results in a proof system similar to that in [MP87].

Proofs for LTS's with fairness: So far, we have only considered LTS's without fairness constraints. Fairness constraints, such as weak or strong fairness on actions, are sometimes required to rule out undesired computations. Manna and Pnueli [MP87] observe that there are two possible ways of handling fairness: one can either incorporate the fairness constraints into the property, or incorporate them into the proof system. They point out that these approaches are closely related. Indeed, the modified proof system corresponds to a particular way of proving the modified property using the original proof system. Therefore, we prefer to keep the simplicity of the proof system, and incorporate any fairness constraints into the property.

4 Proof Generation and Checking

The completeness proof in Theorem 3 shows how to generate a proof for a successful model checking attempt. Such proofs can be generated both by explicit-state and symbolic model checkers. For symbolic model checkers, the invariant assertions are represented by formulas (i.e., BDD's), and it is desirable also for the rank functions to be converted to predicates; i.e., to represent the terms $(\rho_q = k)$ and $(\rho_{q_1} \lhd_q k)$ as the predicates $\rho_=(q,k)$ and $\rho_\lhd(q_1,q,k)$, respectively. Individual proof steps become validity assertions in the assertion language which, for a finite-state model checker, is propositional logic. It is possible for the proof generator and the proof checker to use different symbolic representations and different validity checking methods. For instance, the model checker can be based on BDD methods, while the proof checker represents formulas with syntax trees and utilizes a SAT solver to check validity.

Since we use alternating automata to specify properties, the automaton that defines $\neg f$ can be obtained easily by dualizing A_f: exchanging *true* and *false*, $\wedge$ and $\vee$, and $\langle a \rangle$ and $[a]$ in the transition relation and replacing the parity condition F with its negation $(\emptyset, F_0, \ldots, F_{2n})$. A winning strategy for player I with the dual automaton is a winning strategy for player II with the original automaton. Thus, the set $\mathcal{W}_{II} = \neg \mathcal{W}_I$ can be used to construct a proof of $\neg f$ relative to the dual automaton. To avoid doing extra work to create a proof for $\neg f$ on failure, it is desirable to record approximations for both the μ and ν variables while evaluating $\mathcal{W}_I$: if f holds, the approximations for the μ variables are used to calculate the rank function; if not, a dual proof can be constructed for $\neg f$, using the negations of the approximations recorded for the ν variables of $\mathcal{W}_I$, which are the μ variables in $\mathcal{W}_{II}$. This strategy is followed in our prototype proof generator for COSPAN.

Example: To illustrate the proof generation process, consider the following program and property. All transitions of the program are labeled with τ.

Program $M(m : \mathbb{N})$ (* circular counter *)
var $c : (0..2^m - 1)$; initially $c = 0$; transition $c' = (c + 1)\ mod\ 2^m$

Property A (* $\mathsf{AGF}(c = 0)$, i.e., on all paths, $c = 0$ holds infinitely often *)
states $= \{q_0, q_1\}$; initially q_0;
transition $\delta(q_0, true) = [\tau]q_1, \delta(q_1, c = 0) = q_0, \delta(q_1, c \neq 0) = [\tau]q_1$
parity condition (F_0, F_1), where $F_0 = \{q_0\}, F_1 = \{q_1\}$.

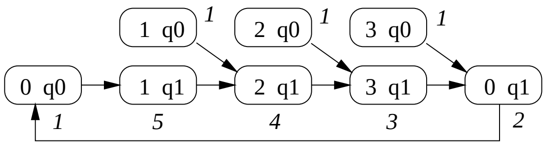

Fig. 1. The graph of $M \times A$ for $m = 2$.

The $\mathcal{W}_I$ formula, as defined in Section 2.2, simplifies to the following, since every state of $M \times A$ is a II-state: $\mathcal{W}_I = (\nu Z_0 : (\mu Z_1 : \Phi_I(Z_0, Z_1))$, where $\Phi_I(Z_0, Z_1) = ((q_0 \Rightarrow [\tau]Z_0) \wedge (q_1 \Rightarrow [\tau]Z_1))$. This formula evaluates to *true* on $M \times A$. Thus, ϕ_{q_0} and ϕ_{q_1}, as calculated in the proof of Theorem 3, are both *true* (i.e., $c \in \{0..2^m - 1\}$). The rank function is calculated by computing the signatures of states satisfying $\mathcal{W}_I$. As there is a single odd index in the parity condition, the signature is a singleton vector, which may be represented by a number. By the definition in Section 2.1, the signature of a state satisfying

$\mathcal{W}_I$ is the smallest index i for which the state belongs to $(\mu^i Z_1 : \Phi_I(\mathcal{W}_I, Z_1))$. This formula simplifies to $(\mu^i Z_1 : (q_0 \vee [\tau]Z_1))$, which essentially calculates the distance to the q_0 state.

The italicized number next to each state in Figure 1 shows its rank. The rank functions are, therefore, $\rho_{q_0}(c) = 1$ and $\rho_{q_1}(c) = \mathit{if}\ (c = 0)\ \mathit{then}\ 2\ \mathit{else}\ (6-c)$. By construction, the Consistency and Initiality properties of the proof are satisfied. Instantiating the general proof scheme of Section 3, the Invariance and Progress obligations reduce to the following, all of which can be seen to hold.

- $[\phi_{q_0}(c) \wedge \mathit{true} \wedge (\rho_{q_0}(c) = k) \Rightarrow [\tau](\phi_{q_1}(c))]$
- $[\phi_{q_1}(c) \wedge (c = 0) \wedge (\rho_{q_1}(c) = k) \Rightarrow (\phi_{q_0}(c) \wedge (\rho_{q_0}(c) < k))]$, and
- $[\phi_{q_1}(c) \wedge (c \neq 0) \wedge (\rho_{q_1}(c) = k) \Rightarrow [\tau](\phi_{q_1}(c) \wedge (\rho_{q_1}(c) < k))]$

Proofs vs. Counterexamples: A natural question that arises concerns the relationship between a proof for $\neg f$ and a counterexample computation for f. This is elucidated in the theorem below.

Theorem 4. For a program M and a linear time, co-Büchi $\forall$-automaton A, if M does not satisfy A, and $M \times A$ has m bits and a counterexample of length n, it is possible to construct a proof for $\neg A$ that needs $2mn$ bits. On the other hand, a proof can be exponentially more succinct than any counterexample.

Proof. In general, a counterexample consists of a path to an accepting state, and a cycle passing through that state. Define the invariants ϕ_q so that they hold only of the states on the counterexample, and let the rank function measure the distance along the counterexample to an accepting state. This can be represented by BDD's of size $2mn$.

On the other hand, consider the program in Figure 1, and the property $\mathsf{G}(c' > c)$. This is false only at $c = 2^m - 1$, so the shortest counterexample has length $2^m + 1$. We can, however, prove failure by defining the invariant to be *true* (really, $\mathsf{EF}(c' \leq c)$), and by letting state c have rank k iff $c + k = 2^m - 1$. This rank function measures the distance to the violating transition. It can be represented by a BDD of size linear in m by interleaving the bits for c and k. Thus, the proof has size linear in m and is, therefore, exponentially more succinct than the counterexample. $\square$

5 Applications

The ability to generate proofs which justify the outcome of model checking makes possible several interesting applications for a certifying model checker.

1. Generating Proofs vs. Generating Counterexamples: We have shown how a certifying model checker can produce a proof of property f upon success and a proof for $\neg f$ on failure. Both types of proofs offer insight on *why* the property succeeds (or fails) to hold of the program. Inspecting success proofs closely may help uncover vacuous justifications or lack of appropriate coverage. The generated proof for $\neg f$ is a compact representation of *all* counterexample computations. This proof can be "unfolded" interactively along the lines of the strategy description in the soundness proof of Theorem 2. This process allows the exploration of various counterexamples *without* having to perform multiple model checking runs (cf. [SS98]).

2. Detecting errors in a model checker: The proof produced by a certifying model checker stands by itself; i.e., it can be checked for correctness independently of the model checker. For instance, the model checker may use BDD's, but the proof can be checked using a SAT solver. It is possible, therefore, to detect errors in the model checker[1]. For instance, if the model checker declares success but produces an erroneous proof, this may be due to a mistake in the implementation which results in a part of the state space being overlooked during model checking.

3. Integrating model checking with theorem proving: Efforts to integrate model checking with theorem proving [JS93,RSS95] have added such a capability at a shallow level, where the result of model checking is accepted as an axiom by the theorem prover. This has been addressed in [YL97,Spr98], where tableau proofs generated using explicit state model checkers are imported into theorem provers. Our proof generation procedure allows symbolic proofs, which are more compact than explicit state proofs, to be used for the same purpose.

Theorem proving, in one form or another, has been used to design and verify abstractions of infinite state systems (cf. [MNS99]), to prove conditions for sound compositional reasoning (cf. [McM99]), and to prove parameterized systems correct (cf. [BBC$^+$00]). In the first two cases, model checking is applied to small subgoals. Proofs generated by a certifying model checker for these subgoals can be composed with the other proofs to produce a single, mechanically checkable, proof script. In the last case, the model checker can be used to produce proofs about small instances of parameterized systems. The shape of the invariance and progress assertions in these proofs can often suggest the assertions needed for proving the general case, which is handled entirely with the theorem prover. This approach has been applied in [PRZ01,APR$^+$01] to invariance properties.

4. Proof Carrying Code: A certifying model checker can produce proofs for arbitrary temporal properties. These proofs can be used with the "proof-carrying-code" paradigm introduced in [NL96] for mobile code: a code producer sends code together with a generated correctness proof, which is checked by the code consumer. The proof generator in [NL98] is tailored to checking memory and type safety. Using a certifying model checker, one can, in principle, generate proofs of arbitrary safety and liveness properties, which would be useful for mobile protocol code.

6 Conclusions and Related Work

There is prior work on automatically generating explicit state proofs for properties expressed in the mu-calculus and other logics, but the proof system and the

[1] So a certifying model checker can be used to "certify" itself!

algorithm of this paper appear to be the first to do so for symbolic representations. In [Kic,YL97], algorithms are given to create tableau proofs in the style of [SW89]. In parallel with our work, Peled and Zuck [PZ01] have developed an algorithm for automatically generating explicit state proofs for LTL properties. The game playing algorithm of [SS98] implicitly generates a kind of proof.

Explicit state proofs are of reasonable size only for programs with small state spaces. For larger programs, symbolic representations are to be preferred, as they result in proofs that are more compact. While the tableau proof system has been extended to symbolic representations in [BS92], the extension requires an external, global termination proof. In contrast, our proof system embeds the termination requirements as locally checkable assertions in the proof.

The proof system presented here is closely related to those of [MP87] (for ∀-automata) and [FG96] (for fair-CTL), but generalizes both systems. The proof system is specifically designed to be locally checkable, so that proofs can be checked easily and mechanically. For some applications, it will be necessary to add rules such as modus ponens to make the proofs more "human-friendly". As we have discussed, though, there are many possible applications for proofs that are generated and checked mechanically, which opens up new and interesting areas for the application of model checking techniques.

Acknowledgements: I would like to thank the members of the formal methods group at Bell Labs, and Hana Chockler, Dave Naumann and Richard Trefler for many interesting discussions on this topic.

References

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, 2001.

[BBC+00] N. Bjorner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 2000.

[BS92] J. Bradfield and C. Stirling. Local model checking for infinite state spaces. *TCS*, 96, 1992.

[CE81] E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*, 1981.

[CM88] K.M. Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Coo78] S.A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput*, 1978.

[EJ91] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS*, 1991.

[EJS93] E. Allen Emerson, C.S. Jutla, and A.P. Sistla. On model-checking for fragments of μ-calculus. In *CAV*, 1993.

[EL86] E.A. and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, 1986.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics*. North-Holland Pub. Co./MIT Press, 1990.

[FG96] L. Fix and O. Grumberg. Verification of temporal properties. *Journal of Logic and Computation*, 1996.

[Flo67] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science XIX*. American Mathemetical Society, 1967.

[HHK96] R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *CAV*, volume 1102 of *LNCS*, 1996.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[JS93] J.J. Joyce and C-J.H. Seger. The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. In *HUG*, volume 780 of *LNCS*, 1993.

[JW95] D. Janin and I. Walukiewicz. Automata for the modal mu-calulus and related results. In *MFCS*, 1995.

[Kel76] R.M. Keller. Formal verification of parallel programs. *CACM*, 1976.

[Kic] A. Kick. Generation of witnesses for global mu-calculus model checking. available at http://liinwww.ira.uka.de/~kick.

[Koz82] D. Kozen. Results on the propositional mu-calculus. In *ICALP*, 1982.

[McM99] K.L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, 1999.

[MNS99] P. Manolios, K.S. Namjoshi, and R. Summers. Linking theorem proving and model-checking with well-founded bisimulation. In *CAV*, 1999.

[MP83] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *POPL*, 1983.

[MP87] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by $\forall$-automata. In *POPL*, 1987.

[NL96] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.

[NL98] G.C. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI*, 1998.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, volume 2031 of *LNCS*, 2001.

[PZ01] D. Peled and L. Zuck. From model checking to a temporal proof. In *The 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, 2001.

[QS82] J-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, 1982.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In *CAV*, volume 939 of *LNCS*, 1995.

[SE84] R.S. Streett and E.A. Emerson. The propositional mu-calculus is elementary. In *ICALP*, 1984. Full version in *Information and Computation* 81(3): 249-264, 1989.

[Spr98] C. Sprenger. A verified model checker for the modal μ-calculus in Coq. In *TACAS*, volume 1384 of *LNCS*, 1998.

[SS98] P. Stevens and C. Stirling. Practical model-checking using games. In *TACAS*, 1998.

[SW89] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. In *TAPSOFT*, 1989. Full version in TCS vol.89, 1991.

[YL97] S. Yu and Z. Luo. Implementing a model checker for LEGO. In *FME*, volume 1313 of *LNCS*, 1997.