In *Proc. 25th Int. Colloquium on Automata, Languages, and Programming.* ICALP-98, Vol. 1443 of LNCS, pp. 1–16, Springer Verlag, 1998.

Algorithmic Verification of Linear Temporal Logic Specifications *

Yonit Kesten† Amir Pnueli‡ Li-on Raviv‡

November 16, 1998

Abstract

In this methodological paper we present a coherent framework for symbolic model checking verification of linear-time temporal logic (LTL) properties of reactive systems, taking full fairness into consideration. We use the computational model of a *fair Kripke structure* (FKS) which takes into account both *justice* (weak fairness) and *compassion* (strong fairness). The approach presented here reduces the model checking problem into the question of whether a given FKS is *feasible* (i.e. has at least one computation).

The contribution of the paper is twofold: On the methodological level, it presents a direct self-contained exposition of full LTL symbolic model checking without resorting to reductions to either CTL or automata. On the technical level, it extends previous methods by dealing with compassion at the algorithmic level instead of adding it to the specification, and providing the first symbolic method for checking feasibility of FKS's (equivalently, symbolically checking for the emptiness of Streett automata).

The presented algorithms can also be used (with minor modifications) for symbolic model-checking of CTL formulas over fair Kripke structures with compassion requirements.

1 Introduction

Two brands of temporal logics have been proposed over the years for specifying the properties of reactive systems: the *linear time* brand LTL [GPSS80] and the *branching time* variant CTL [CE81]. Also two methods for the formal verification of the temporal properties of reactive systems have been developed: the *deductive approach* based on interactive theorem proving, and the fully automatic *algorithmic approach*, widely known as *model checking*. Tracing the evolution of these ideas, we find that the deductive approach adopted LTL as its main vehicle for specification, while the model-checking approach used CTL as the specification language [CE81], [QS82].

This is more than a historical coincidence or a matter of personal preference. The main advantage of CTL for model checking is that it is *state-based* and, therefore, the process of verification can be performed by straightforward *labeling* of the existing states in the Kripke structure, leading to no further expansion or unwinding of the structure. In contrast, LTL is *path-based* and, since many paths can pass through a single state, labeling a structure by the LTL sub-formulas it satisfies necessarily requires splitting the state into several copies. This is the reason why the development

*This research was supported in part by an infra-structure grant from the Israeli Ministry of Science and Art, a grant from the U.S.-Israel Binational Science Foundation, and a gift from Intel.

†Dept. of Communication System Engineering, Ben Gurion University, `ykesten@bgumail.bgu.ac.il` — Contact author

‡Weizmann Institute of Science, {`amir, lion`}`@wisdom.weizmann.ac.il`

of model-checking algorithms for LTL always lagged several years behind their first introduction for the CTL logic.

The first model-checking algorithms were based on the enumerative approach, constructing an explicit representation of all reachable states of the considered system [CE81], and were developed for the branching-time temporal logic CTL. The LTL version of these algorithms was developed in [LP85] for the future fragment of propositional LTL (PTL), and extended in [LPZ85] to the full PTL. The basic fixed-point computation algorithm for the identification of fair computations presented in [LP85], was developed independently in [EL85] for FCTL (fair CTL). Observing that upgrading from justice to full fairness (i.e., adding compassion) is reflected in the automata view of verification as an upgrade from a Buchi to a Street automaton, we can view the algorithms presented in [EL85] and [LP85] as algorithms for checking the emptiness of Street automata [VW86]. An improved algorithm solving the related problem of emptiness of Street automata, was later presented in [HT96]. The development of the impressively efficient symbolic verification methods and their application to CTL [BCM+92] raised the question whether a similar approach can be applied to PTL. The first satisfactory answer to this question was given in [CGH94], which showed how to reduce model checking of a future PTL formula into CTL model checking. The advantages of this approach is that, following a preliminary transformation of the PTL formula and the given system, the algorithm proceeds by using available and efficient CTL model checkers such as SMV.

A certain weakness of all the available symbolic model checkers is that, in their representation of fairness, they only consider the concept of *justice* (weak fairness). As suggested by many researchers, another important fairness requirement is that of *compassion* (strong fairness) (e.g., [GPSS80], [LPS81], [Fra86]). This type of fairness is particularly useful in the analysis of systems that use semaphores, synchronous communication, and other special coordination primitives. A partial answer to this criticism is that, since compassion can be expressed in LTL (but not in CTL), once we developed a model-checking method for LTL, we can always add the compassion requirements as an antecedent to the property we wish to verify. A similar answer is standardly given for symbolic model checkers that use the μ-calculus as their specification language, because compassion can also be expressed as a μ-calculus formula [SdRG89]. The only question remaining is how practical this is.

In this methodological paper (summarizing an invited talk), we present an approach to the symbolic model checking of LTL formulas, which takes into account full fairness, including both justice and compassion. The presentation of the approach is self-contained and does not depend on a reduction to either CTL model checking (as in [CGH94]) or to automata. The treatment of the LTL component is essentially that of a symbolic construction of a tableau by assigning a new auxiliary variable to each temporal sub-formula of the property we wish to verify. In that, our approach resembles very much the reduction method used in [CGH94] which, in turn, is an extension of the *statification* method used in [MP91a] and [MP95] to deal with the past fragment of LTL.

Another work related to the approach developed here is presented in [HKSV97], where a BDD-based symbolic algorithm for bad cycle detection is presented. This algorithm solves the problem of finding all those cycles within the computation graph, which satisfy some fairness constraints. However, the algorithm of [HKSV97] deals only with *justice*, and does not deal with *compassion*. According to the automata-theoretic view, [HKSV97] presents a symbolic algorithm for the problem of emptiness of Buchi automata while the algorithms presented here, provide a symbolic solution to the emptiness problem of Street automata.

The symbolic model-checking algorithms presented here are not restricted to the treatment of LTL formulas. With minor modifications they can be applied to check the CTL formula $\mathbf{E}_{fair}\mathbf{G}p$, where the *fair* subscript refers now to full fairness.

2 Fair Kripke Structure

As a computational model for reactive systems, we take the model of *fair kripke structure* (FKS). Such a system $K : \langle V, \Theta, \rho, \mathcal{J}_K, \mathcal{C}_K \rangle$ consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of typed *state variables.* For the case of finite-state systems, we assume that all state variables range over finite domains. We define a *state* s to be a type-consistent interpretation of V, assigning to each variable $u \in V$ a value $s[u]$ in its domain. We denote by Σ the set of all states.

- Θ : The *initial condition.* This is an assertion characterizing all the initial states of an FKS. A state is defined to be *initial* if it satisfies Θ.

- ρ : A *transition relation.* This is an assertion $\rho_\tau(V, V')$, relating a state $s \in \Sigma$ to its K-successor $s' \in \Sigma$ by referring to both unprimed and primed versions of the state variables. An unprimed version of a state variable refers to its value in s, while a primed version of the same variable refers to its value in s'. For example, the transition relation $x' = x + 1$ asserts that the value of x in s' is greater by 1 than its value in s.

- $\mathcal{J}_K = \{J_1, \ldots, J_k\}$: A set of *justice* requirements (also called *weak fairness requirements*). Intuitively, the justice requirement $J \in \mathcal{J}_K$ stipulates that every computation contains infinitely many J-state (states satisfying J).

- $\mathcal{C}_K = \{\langle p_1, q_1 \rangle, \ldots \langle p_n, q_n \rangle\}$: A set of *compassion* requirements (also called *strong fairness requirements*). Intuitively, the compassion requirement for $\langle p, q \rangle \in \mathcal{C}_K$ stipulates that every computation containing infinitely many p-states also contains infinitely many q-states.

The transition relation $\rho(V, V')$ identifies state s' as a *K-successor* of state s if

$$\langle s, s' \rangle \models \rho(V, V'),$$

where $\langle s, s' \rangle$ is the joint interpretation which interprets $x \in V$ as $s[x]$, and interprets x' as $s'[x]$.

Let $\sigma : s_0, s_1, s_2, ...,$ be an infinite sequence of states, φ be an assertion (state formula), and let $j \geq 0$ be a natural number. We say that j is a φ-position of σ if s_j is a φ-state.

Let K be an FKS for which the above components have been identified. We define a *computation* of K to be an infinite sequence of states $\sigma : s_0, s_1, s_2, ...,$ satisfying the following requirements:

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For each $j = 0, 1, ...,$ the state s_{j+1} is a K-successor of the state s_j.
- *Justice:* For each $J \in \mathcal{J}_K$, σ contains infinitely many J-positions
- *Compassion:* For each $\langle p, q \rangle \in \mathcal{C}_K$, if σ contains infinitely many p-positions, it must also contain infinitely many q-positions.

For an FKS K, we denote by $\mathcal{C}omp(K)$ the set of all computations of K.

3 Parallel Composition of FKS's

Fair Kripke structures can be composed in parallel. Let $K_1 = \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle$ and $K_2 = \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle$ be two fair Kripke structures. We consider two versions of parallel composition.

3.1 Asynchronous Parallel Composition

We define the *asynchronous parallel composition* of two FKS's to be

$$\langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle \quad = \quad \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle \parallel \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle,$$

where,

$$\begin{array}{rcl} V & = & V_1 \cup V_2 \\ \Theta & = & \Theta_1 \wedge \Theta_2 \\ \rho & = & (\rho_1 \wedge pres(V_2 - V_1)) \ \vee \ (\rho_2 \wedge pres(V_1 - V_2)) \\ \mathcal{J} & = & \mathcal{J}_1 \cup \mathcal{J}_2 \\ \mathcal{C} & = & \mathcal{C}_1 \cup \mathcal{C}_2. \end{array}$$

The asynchronous parallel composition of systems K_1 and K_2 is a new system K whose basic actions are chosen from the basic actions of its components, i.e., K_1 and K_2. Thus, we can view the execution of K as the *interleaved execution* of K_1 and K_2, and can use asynchronous composition. in order to construct big concurrent systems from smaller components.

As seen from the definition, K_1 and K_2 may have different as well as common state variables, and the variables of K are the union of all of these variables. The initial condition of K is the conjunction of the initial conditions of K_1 and K_2. The transition relation of K states that at any step, we may choose to perform a step of K_1 or a step of K_2. However, when we select one of the two systems, we should also take care to preserve the private variables of the other system. For example, choosing to execute a step of K_1, we should preserve all variables in $V_2 - V_1$.

The justice and compassion sets of K are formed as the respective unions of the justice and compassion sets of the component systems.

3.2 Synchronous Parallel Composition

We define the *synchronous parallel composition* of two FKS's to be

$$\langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle \quad = \quad \langle V_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1 \rangle \parallel\!| \langle V_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2 \rangle,$$

where,

$$\begin{array}{rcl} V & = & V_1 \cup V_2 \\ \Theta & = & \Theta_1 \wedge \Theta_2 \\ \rho & = & \rho_1 \ \wedge \ \rho_2 \\ \mathcal{J} & = & \mathcal{J}_1 \cup \mathcal{J}_2 \\ \mathcal{C} & = & \mathcal{C}_1 \cup \mathcal{C}_2. \end{array}$$

The synchronous parallel composition of systems K_1 and K_2 is a new system K, each of whose basic actions consists of the joint execution of an action of K_1 and an action of K_2. Thus, we can view the execution of K as the *joint execution* of K_1 and K_2.

In some cases, in particular when considering hardware designs which are naturally synchronous, we may also use synchronous composition to assemble a system from its components. However, our primary use of synchronous composition is for combining a system with a *tester* for a temporal property (described in Section 5) which continuously monitor the behavior of the system and judges whether the system satisfies the desired property.

4 Linear Temporal Logic

As a requirement specification language for reactive systems we take the propositional fragment of *linear temporal logic* [MP91b].

Let $\mathcal{P}$ be a finite set of propositions. A *state formula* is constructed out of propositions and the boolean operators $\neg$ and $\vee$. A *temporal formula* is constructed out of state formulas to which we apply the boolean operators and the following basic temporal operators:

$\bigcirc$ – Next $\ominus$ – Previous
$\mathcal{U}$ – Until $\mathcal{S}$ – Since

A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, ...,$ where each state s_j provides an interpretation for the variables mentioned in p.

Given a model σ, as above, we present an inductive definition for the notion of a temporal formula p holding at a position $j \geq 0$ in σ, denoted by $(\sigma, j) \models p$.

- For a state formula p,
  $(\sigma, j) \models p \iff s_j \models p$
  That is, we evaluate p locally, using the interpretation given by s_j.
- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \iff (\sigma, j+1) \models p$
- $(\sigma, j) \models p \mathcal{U} q \iff$ for some $k \geq j, (\sigma, k) \models q$,
  and for every i such that $j \leq i < k, (\sigma, i) \models p$
- $(\sigma, j) \models \ominus p \iff j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p \mathcal{S} q \iff$ for some $k \leq j, (\sigma, k) \models q$,
  and for every i such that $j \geq i > k, (\sigma, i) \models p$

We refer to the set of variables that occur in a formula p as the *vocabulary of* p. For a state formula p and a state s such that p holds on s, we say that s is a *p-state*.

If $(\sigma, 0) \models p$, we say that p *holds* on σ, and denote it by $\sigma \models p$. A formula p is called *satisfiable* if it holds on some model. A formula is called *temporally valid* if it holds on all models.

The notion of validity requires that the formula holds over *all* models. Given an FKS K, we can restrict our attention to the set of models which correspond to computations of K, i.e., $\mathcal{C}omp(K)$. This leads to the notion of K-validity, by which a temporal formula p is *K-valid* (valid over FKS K) if it holds over all the computations of P. Obviously, any formula that is (generally) valid is also K-valid for any FKS K. In a similar way, we obtain the notion of K-satisfiability.

5 Construction of Testers for Temporal Formulas

In this section, we present the construction of a *tester* for a PTL formula φ, which is an FKS T_φ characterizing all the sequences which satisfy φ.

For a formula ψ, we write $\psi \in \varphi$ to denote that ψ is a sub-formula of (possibly equal to) φ. Formula ψ is called *principally temporal* if its main operator is a temporal operator.

The FKS T_φ is given by

$$T_\varphi\colon \quad \left\langle V_\varphi,\, \Theta_\varphi,\, \rho_\varphi, \mathcal{J}_\varphi, \mathcal{C}_\varphi \right\rangle,$$

where the components are specified as follows:

System Variables

The system variables of T_φ consist of the vocabulary of φ plus a set of auxiliary boolean variables

$$X_\varphi: \quad \{x_p \mid p \in \varphi \text{ a principally temporal sub-formula of } \varphi\},$$

which includes an auxiliary variable x_p for every p, a principally temporal sub-formula of φ. The auxiliary variable x_p is intended to be true in a state of a computation iff the temporal formula p holds at that state.

We define a mapping χ which maps every sub-formula of φ into an assertion over V_φ.

$$\chi(\psi) = \begin{cases} \psi & \text{for } \psi \text{ a state formula} \\ \neg\chi(p) & \text{for } \psi = \neg p \\ \chi(p) \vee \chi(q) & \text{for } \psi = p \vee q \\ x_\psi & \text{for } \psi \text{ a principally temporal formula} \end{cases}$$

The mapping χ distributes over all boolean operators. When applied to a state formula it yields the formula itself. When applied to a principally temporal sub-formula p it yields x_p.

Initial Condition

The initial condition of T_φ is given by

$$\Theta_\varphi: \quad \chi(\varphi) \;\wedge \bigwedge_{\ominus p \in \varphi} \neg x_{\ominus p} \;\wedge \bigwedge_{p \mathcal{S} q \in \varphi} (x_{p\mathcal{S}q} \;\leftrightarrow\; \chi(q)).$$

Thus, the initial condition requires that all initial states satisfy $\chi(\varphi)$, and that all auxiliary variables encoding "Previous" formulas are initially false. This corresponds to the observation that all formulas of the form $\ominus p$ are false at the first state of any sequence. In addition, Θ_φ requires that the truth value of $x_{p\mathcal{S}q}$ equals the truth value of $\chi(q)$, corresponding to the observation that the only way to satisfy the formula $p\,\mathcal{S}q$ at the first state of a sequence is by satisiying q.

Transition Relation

The transition relation of T_φ is given by

$$\rho_\varphi: \quad \left(\begin{array}{l} \wedge \displaystyle\bigwedge_{\ominus p \in \varphi} (x'_{\ominus p} \;\leftrightarrow\; \chi(p)) \;\;\wedge \bigwedge_{p\mathcal{S}q \in \varphi} (x'_{p\mathcal{S}q} \;\leftrightarrow\; (\chi'(q) \vee (\chi'(p) \wedge x_{p\mathcal{S}q}))) \\ \wedge \displaystyle\bigwedge_{\bigcirc p \in \varphi} (x_{\bigcirc p} \;\leftrightarrow\; \chi'(p)) \;\;\wedge \bigwedge_{p\mathcal{U}q \in \varphi} (x_{p\mathcal{U}q} \;\leftrightarrow\; (\chi(q) \vee (\chi(p) \wedge x'_{p\mathcal{U}q}))) \end{array}\right)$$

Note that we use the form x_ψ when we know that ψ is principally temporal and the form $\chi(\psi)$ in all other cases. The expression $\chi'(\psi)$ denotes the primed version of $\chi(p)$. The conjuncts of the transition relation corresponding to the *Since* and the *Until* operators are based on the following expansion formulas:

$$p\,\mathcal{S}q \iff q \vee (p \wedge \ominus(p\,\mathcal{S}q)) \qquad\qquad p\,\mathcal{U}q \iff q \vee (p \wedge \bigcirc(p\,\mathcal{U}q))$$

Fairness Requirements

The justice set of T_φ is given by

$$\mathcal{J}_\varphi: \quad \{\chi(q) \vee \neg x_{p\mathcal{U}q} \mid p\mathcal{U}q \in \varphi\}.$$

Thus, we include in $\mathcal{J}_\varphi$ the disjunction $\chi(q) \vee \neg x_{p\mathcal{U}q}$ for every *until* formula $p\mathcal{U}q$ which is a subformula of φ. The justice requirement for the formula $p\mathcal{U}q$ ensures that the sequence contains infinitely many states at which $\chi(q)$ is true, or infinitely many states at which $x_{p\mathcal{U}q}$ is false.

The compassion set of T_φ is always empty.

Correctness of the Construction

For a set of variables U, we say that sequence $\widetilde{\sigma}$ is a *U-variant* of sequence σ if σ and $\widetilde{\sigma}$ agree on the interpretation of all variables, except possibly the variables in U.

The following claim states that the construction of the tester T_φ correctly captures the set of sequences satisfying the formula φ.

Claim 1 *A state sequence σ satisfies the temporal formula φ iff σ is an X_φ-variant of a computation of T_φ.*

6 Checking for Feasibility

An FKS $K : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ is called *feasible* if it has at least one computation. In this section we present a symbolic algorithm for checking feasibility of a finite-state FKS.

We define a *run* of K to be a finite or infinite sequence of states which satisfies the requirements of initiality and consecution but not necessarily any of the justice or compassion requirements. We say that a state s is *K-accessible* if it appears in some run of K. When K is understood from the context, we simply say that state s is accessible.

The symbolic algorithm presented here, is inspired by the full state-enumeration algorithm originally presented in [LP85] and [EL85] (for full explanations and proofs see [Lic91] and [MP95]). The enumerative algorithm constructs a *state-transition graph* G_K for K. This is a directed graph whose nodes are all the K-accessible states, and whose edges connect node s to node s' iff s' is a K-successor of s. If system K has a computation it corresponds to an infinite path in the graph G_K which starts at a K-initial state. We refer to such paths as *initialized paths*.

Subgraphs of G_K can be specified by identifying a subset $S \subseteq G_K$ of the nodes of G_K. It is implied that as the edges of the subgraph we take all the original G_K-edges connecting nodes (states) of S.

A subgraph S is called *just* if it contains a J-state for every justice requirement $J \in \mathcal{J}$. The subgraph S is called *compassionate* if, for every compassion requirement $(p, q) \in \mathcal{C}$, S contains a q-state, or S does not contain any p-state. A subgraph is *singular* if it is composed of a single state which is not connected to itself. A subgraph S is *fair* if it is a non-singular strongly connected subgraph which is both just and compassionate.

For π, an infinite initialized path in G_K, we denote by $\mathit{Inf}(\pi)$ the set of states which appear infinitely many times in π. The following claims, which are proved in [Lic91], connect computations of K with fair subgraphs of G_K.

Claim 2 *The infinite initialized path π is a computation of K iff* $\mathrm{Inf}(\pi)$ *is a fair subgraph of G_K.*

Corollary 3 *A system K is feasible iff G_K contains a fair subgraph.*

The Symbolic algorithm

The symbolic algorithm, aimed at exploiting the data structure of OBDD's, is presented in a general set notation.

Let Σ denote the set of all states of an FKS K. A *predicate* over Σ is any subset $U \subseteq \Sigma$. A (binary) *relation* over Σ is any set of pairs $R \subseteq \Sigma \times \Sigma$. Since both predicates and relations are sets, we can freely apply the set-operations of union, intersection, and complementation to these objects. In addition, we define two operations of composition of predicates and relations. For a predicate U and relation R, we define the operations of pre- and post-composition as follows:

$$\begin{array}{rcl} R \circ U & = & \{s \in \Sigma \mid (s, s') \in R \text{ for some } s' \in U\} \\ U \circ R & = & \{s \in \Sigma \mid (s_0, s) \in R \text{ for some } s_0 \in U\} \end{array}$$

If we view R as a transition relation, then $R \circ U$ is the set of all R-predecessors of U-states, and $U \circ R$ is the set of all R-successors of U-states.

To capture the set of all states that can reach a U-state in a finite number of R-steps (including zero), we define

$$R^* \circ U \quad = \quad U \ \cup\ R \circ U \ \cup\ R \circ (R \circ U) \ \cup\ R \circ (R \circ (R \circ U)) \ \cup\ \cdots.$$

It is easy to see that $R^* \circ U$ converges after a finite number of steps. In a similar way, we define

$$U \circ R^* \quad = \quad U \ \cup\ U \circ R \ \cup\ (U \circ R) \circ R \ \cup\ ((U \circ R) \circ R) \circ R \ \cup\ \cdots,$$

which captures the set of all states reachable in a finite number of R-steps from a U-state.

For predicates U and W, we define the relation $U \times W$ as

$$U \times W \quad = \quad \{(s_1, s_2) \in \Sigma^2 \mid s_1 \in U, s_2 \in W\}.$$

For an assertion φ over V_K (the system variables of FKS K), we denote by $\|\varphi\|$ the predicate consisting of all states satisfying φ. Similarly, for an assertion ρ over (V_K, V'_K), we denote by $\|\rho\|$ the relation consisting of all state pairs $\langle s, s' \rangle$ satisfying ρ.

Algorithm FEASIBLE (K) : **predicate** — Check feasibility of an FKS

$\quad$ new, old : **predicate**
$\quad$ R : **relation**

$old := \emptyset$
$R := \|\rho_K\|$
$new := \|\Theta_K\| \circ R^*$
while $(new \neq old)$ **do**
begin
$\quad$ $old := new$
$\quad$ **for each** $J \in \mathcal{J}$ **do**
$\quad\quad$ $new := (new \cap \|J\|) \circ R^*$
$\quad$ **for each** $(p, q) \in \mathcal{C}$ **do**
$\quad$ **begin**
$\quad\quad$ $new := (new - \|p\|) \ \cup\ (new \cap \|q\|) \circ R^*$
$\quad\quad$ $R := R \cap (\Sigma \times new)$
$\quad$ **end**
$\quad$ **while** $(new \neq new \cap (new \circ R))$ **do**
$\quad\quad$ $new := new \cap (new \circ R)$
end
return(new)

The algorithm consists of a main loop which converges when the values of the predicate variable *new* coincide on two successive visits to line 4. Prior to entry to the main loop we place in R the transition relation implied by ρ_K, and compute in *new* the set of all accessible states.

The main loop contains three inner loops. The inner loop at lines 6–7 removes from *new* all states which are not R^*-successors of some J-state for all justice requirements $J \in \mathcal{J}$.

The loop at lines 8–10, removes from *new* all p-states which are not R^*-successors of some q-state for some $(p, q) \in \mathcal{C}$. Line 10 restricts again R to pairs (s_1, s_2) where s_2 is currently in *new*.

Finally, the loop at lines 11–12, successively removes from *new* all states which do not have a predecessor in *new*. This process is iterated until all states in the set *new* have a predecessor in the set.

Correctness of the Set-Based Algorithm

Let K be an FKS and U_K be the set of states resulting from the application of algorithm FEASIBLE over K. The following sequence of claims establish the correctness of the algorithm

Claim 4 (Termination) *The algorithm* FEASIBLE *terminates.*

Let us denote by new_i^4 the value of variable *new* on the i'th visit ($i = 0, 1, \ldots$) to line 4 of the algorithm. Since new_0^4 is closed under R-succession, i.e. $new_0^4 \circ R^* = new_0^4$, it is not ddificult to see that $new_1^4 \subseteq new_0^4$. From this, it can be established by induction on i that $new_{i+1}^4 \subseteq new_i^4$, for every $i = 0, 1, \ldots$. It follows that the sequence $|new_0^4| \geq |new_1^4| \geq |new_2^4| \cdots$, is a non-increasing sequence of natural numbers which must eventually stabilize. At the point of stabilization, we have that $new_{i+1}^4 = new_i^4$, implying termination of the algorithm.

Claim 5 (Completeness) *If K is feasible then $U_K \neq \emptyset$.*

Assume that K is feasible. According to Corollary 2, G_K contains a fair subgraph S. By definition, S is a non-singular strongly-connected subgraph which contains a J-state for every $J \in \mathcal{J}$, and such that, for every $(p, q) \in \mathcal{C}$, S contains a q-state or contains no p state. Following the oprtations performed by Algorithm FEASIBLE, we can show that S is contained in the set *new* at all locations beyond the first visit to line 4. This is because any removal of states from *new* which is carried out in lines 7, 9, and 12, cannot remove any state of S. Consequently, S must remain throughout the process and will be contained in U_K, implying the non-emptiness of U_K.

Claim 6 (Soundness) *If $U_K \neq \emptyset$ then K is feasible.*

Assume that U_K is non-empty. Let us characterize the properties of an arbitrary state $s \in U_K$. We know that s is K-accessible. For every $J \in \mathcal{J}$, s is reachable from a J-state by a path fully contained within U_K. For every $(p, q) \in \mathcal{C}$, either s is not a p-state, or s is reachable from a q-state by a U_K-path.

Let us decompose U_K into maximal strongly-connected subgraphs. At least one subgraph S_0 is *initial* in this decomposition, in the sense that every U_K-edge entering an S_0-state also originates at an S_0-state. We argue that S_0 is fair. By definition, it is strongly connected. It cannot be singular, because then it would consist of a single state s that would have been removed on the last execution of the loop at lines 11-12. Let s be an arbitrary state within S_0. For every $J \in \mathcal{J}$, s is reachable from some J-state $\widetilde{s} \in U_K$ by a U_K-path. Since S_0 is initial within U_K, this path must be fully contained within S_0 and, therefore, $\widetilde{s} \in S_0$. In a similar way, we can show that S_0 satisfies all the comapssion requirements. Thus, if $U_K \subseteq G_K$ is non-empty, it contains a fair subgraph which, by Corollary 3, establishes that K is feasible.

Lemmas 5 and 6 lead to the following conclusion:

Corollary 7 *K is feasible iff the set $U_K \neq \emptyset$.*

The original enumerative algorithms of [EL85] and [LP85] were based on recursive exploration of strongly connected subgraphs. Strongly connected subgraphs require closure under both successors and predecessors. As our algorithm (and its proof) show, it is possible to relax the requirement of bi-directional closure into either closure under predecessors and looking for terminal components, or symmetrically requiring closure under successors and looking for initial components which is the approach taken in Algorithm FEASIBLE. This may be an idea worth exploring even in the enumerative case, and to which we can again apply the *lock-step search* optimization described in [HT96].

6.1 How to Model Check?

Having presented an algorithm for checking whether a given FKS is feasible, we outline our proposed algorithm for model checking that an FKS K satisfies a temporal formula φ. The algorithm is based on the following claim:

Claim 8 *$K \models \varphi$ iff $K|||T_{\neg\varphi}$ is not feasible.*

Thus, to check that $K \models \varphi$ we apply algorithm FEASIBLE to the composed FKS $K|||T_{\neg\varphi}$ and declare success if the algorithm finds that $K|||T_{\neg\varphi}$ is infeasible.

7 Extracting a Witness

To use formal verification as an effective debugging tool in the context of verification of finite-state reactive systems checked against temporal properties, a most useful information is a computation of the system which violates the requirement, to which we refer as a *witness*. Since we reduced the problem of checking $K \models \varphi$ to checking the feasibility of $K \mathbin{|||} T_{\neg\varphi}$, such a witness can be provided by a computation of the combined FKS $K \mathbin{|||} T_{\neg\varphi}$.

In the following we present an algorithm which produces a computation of an FKS that has been declared feasible. We introduce the **list** data structure to represent a linear list of states. We use Λ to denote the empty list. For two lists $L_1 = (s_1, \ldots, s_a)$ and $L_2 = (s_a, \ldots, s_b)$, we denote by $L_1 * L_2$ their *fusion*, defined by $L_1 * L_2 \quad = \quad (s_1, \ldots, s_a, \ldots, s_b)$

Finally, for a list L, we denote by $last(L)$ the last element of L. For a non-empty predicate $U \subseteq \Sigma$, we denote by $choose(U)$ a consistent choice of one of the members of U.

The function $path(source, destination, R)$, presented in Fig. 1, returns a list which contains the shortest R-path from a state in *source* to a state in *destination*. In the case that *source* and *destination* have a non-empty intersection, *path* will return a state belonging to this intersection which can be viewed as a path of length zero.

Finally, in figure 2 we present an algorithm which produces a computation of a given FKS. Although a computation is an infinite sequence of states, if K is feasible, it always has an *ultimately periodic* computation of the following form:

$$\sigma: \quad \underbrace{s_0, s_1, \ldots, s_k}_{prefix}, \underbrace{s_{k+1}, \ldots, s_k}_{period}, \underbrace{s_{k+1}, \ldots, s_k}_{period}, \cdots, \underbrace{s_{k+1}, \ldots, s_k}_{period}, \cdots$$

Based on this observation, our witness extracting algorithm will return as result the two finite sequences *prefix* and *period*.

Function $path(source, destination$: **predicate**; R : **relation**) : **list** —
— — Compute shortest path from $source$ to $destination$

$start, f$: **predicate**
L : **list**
s : **state**
$start := source$
$L := \Lambda$
while $(start \cap destination = \emptyset)$ **do**
begin
  $f := R \circ destination$
  while $(start \cap f = \emptyset)$ **do**
    $f := R \circ f$
  $s := choose(start \cap f)$
  $L := L * (s)$
  $start := s \circ R$
end
return $L * (choose(start \cap destination))$

Figure 1: Function *path*.

The algorithm starts by checking whether FKS K is feasible. It uses Algorithm FEASIBLE to perform this check. If K is found to be infeasible, the algorithm exits while providing a pair of empty lists as a result.

If K is found to be feasible, we store in *final* the graph returned by FEASIBLE. This graph contains all the fair SCS's reachable from am initial state. We restrict the transition relation R to depart only from states within *final*. Next, we perform a search for an initial *maximal strongly connected subgraph* (MSCS) within *final*. The search starts at $s \in final$, an arbitrarily chosen state within *final*.

In the loop at lines 5 and 6 we search for a state s satisfying $R^* \circ \{s\} \quad \subseteq \quad \{s\} \circ R^*$. i.e. a state all of whose R^*-predecessors are also R^*-successors. This is done by successively replacing s by a state $s \in R^* \circ \{s\} - \{s\} \circ R^*$ as long as the set of s-predecessors is not contained in the set of s-successors. Eventually, execution of the loop must terminate when s reaches an initial MSCS within *final*. Termination is guaranteed because each such replacement move the state from one MSCS to a preceding MSCS in the canonical decomposition of *final* into MSCS's.

A central point in the proof of correctness of Algorithm FEASIBLE established that any initial MSCS within *final* is a fair subgraph. Line 7 computes the MSCS containing s and assigns it to the variable *final*, while line 8 restricts the transition relation to edges connecting states within *final*. Line 9 draws a (shortest) path from an initial state to the subgraph *final*.

Lines 10 – 17 construct in *period* a traversing path, starting at $last(prefix)$ and returning to the same state, while visiting on the way states that ensure that an infinite repetition of the period will fulfill all the fairness requirements.

Lines 11–13 ensure that *period* contains a J-state, for each $J \in \mathcal{J}$. To prevent unnecessary visits to state, we extend the path to visit the next J-state only if the part of *period* that has already been constructed did not visit any J-state. Lines 14–16 similarly take care of compassion. Here we extend the path to visit a q-state only if the constructed path did not already do so and

Algorithm WITNESS (K) : [**list**, **list**] — Extract a witness for a feasible FKS.

final	:	**predicate**
R	:	**relation**
prefix, *period*	:	**list**
s	:	**state**

1. *final* := FEASIBLE (K)
2. **if** (*final* $= \emptyset$) **then return** (Λ, Λ)
3. $R := \|\rho_K\| \cap (\mathit{final} \times \Sigma)$
4. $s := choose(final)$
5. **while** $(R^* \circ \{s\} - \{s\} \circ R^* \neq \emptyset)$ **do**
6. $s := choose(R^* \circ \{s\} - \{s\} \circ R^*)$
7. $\mathit{final} := R^* \circ \{s\} \ \cap \ \{s\} \circ R^*$
8. $R := R \cap (\mathit{final} \times \mathit{final})$
9. $\mathit{prefix} := \mathit{path}(\|\Theta_K\|, \mathit{final}, \|\rho_K\|)$
10. $\mathit{period} := (\mathit{last}(\mathit{prefix}))$
11. **for each** $J \in \mathcal{J}$ **do**
12.   **if** $(\mathit{list\text{-}to\text{-}set}(\mathit{period}) \cap \|J\| = \emptyset)$ **then**
13. $\mathit{period} := \mathit{period} * \mathit{path}(\{\mathit{last}(\mathit{period})\}, \mathit{final} \cap \|J\|, R)$
14. **for each** $(p, q) \in \mathcal{C}$ **do**
15.   **if** $(\mathit{list\text{-}to\text{-}set}(\mathit{period}) \cap \|q\| = \emptyset \ \wedge \ \mathit{final} \cap \|p\| \neq \emptyset)$ **then**
16. $\mathit{period} := \mathit{period} * \mathit{path}(\{\mathit{last}(\mathit{period})\}, \mathit{final} \cap \|q\|, R)$
17. $\mathit{period} := \mathit{period} * \mathit{path}(\{\mathit{last}(\mathit{period})\}, \{\mathit{last}(\mathit{prefix})\}, R)$
18. **return** $(\mathit{prefix}, \mathit{period})$

Figure 2: Algorithm WITNESS.

the MSCS *final* contains some p-state. Finally, in line 17, we complete the path to form a closed cycle by looping back to $\mathit{last}(\mathit{prefix})$.

8 Implementation and Experimental Results

The algorithms described in the paper have been implemented within the TLV system [PS96]. Since the novel features of the approach concern systems which rely on compassion for their correctness, we chose as natural test cases several programs using semaphores for coordination between processes.

A simple solution to the dining philosophers problem is presented as program DINE in Fig. 3.

While satisfying the safety requirement that no two neighboring philosophers can dine at the same time, this naive algorithm fails to satisfy the liveness requirement of *accessibility* by which every philosopher who wishes to dine will eventually do so.

To guarantee the property of accessibility, we must use better algorithms. However, we prefer to model check the incorrect program DINE in order to test the ability of our algorithms to produce appropriate counter-examples.

In the table of Fig. 4, we present the results of running our verification algorithms checking the property of accessibility over program DINE for different numbers of processes. The numbers are given in seconds of running time over a *Sun* 450.

$$
\begin{array}{lll}
\textbf{in} & n & : \textbf{integer where } n \geq 2 \\
\textbf{local} & c & : \textbf{array } [1..n] \textbf{ where } c = 1
\end{array}
$$

$$
\mathop{\Big\|}_{j=1}^{n} P[j] :: \left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
\ell_1 : & \textbf{noncritical} \\
\ell_2 : & \textbf{request } c[j] \\
\ell_3 : & \textbf{request } c[j \oplus_n 1] \\
\ell_4 : & \textbf{critical} \\
\ell_5 : & \textbf{release } c[j] \\
\ell_6 : & \textbf{release } c[j \oplus_n 1]
\end{array} \right]
\end{array} \right]
$$

Figure 3: Program DINE: a simple solution to the dining philosophers problem.

n	Time to Analyze	Time to produce A witness	Witness size
7	7	19	22
8	15	62	25
9	28	183	28
10	51	513	31
11	93	1227	34
12	174	2695	37
13	303	5589	40

Figure 4: Results for model checking program DINE using the algorithms with builtin compassion.

To examine the efficiency of our approach versus the possibility of including the compassion requirement as part of the property to be verified, we ran the same problem but this time added the compassion requirements to the specification and ran our algorithm with an empty compassion set. The results are summarized in the table of Fig. 5. As can be seen from comparing these tables, the algorithms with the compassion requirements incorporated within the FKS model are far superior to the runs in which the compassion requirements were added to the property to be verified.

References

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer-Verlag, 1981.

[CGH94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In D. L. Dill, editor, *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 415–427. Springer-Verlag, 1994.

n	Time to Analyze	Time to produce A witness	Witness size
3	4	4	11
4	25	25	14
5	133	139	17
6	612	651	20
7	2279	2473	23

Figure 5: Results for model checking program DINE using the algorithms with the compassion requirement added to the property.

[EL85] E.A. Emerson and C.L. Lei. Modalities for model checking: Branching time strikes back. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 84–96, 1985.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, 1986.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symp. Princ. of Prog. Lang.*, pages 163–173, 1980.

[HKSV97] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using BDDs. In O. Grumberg, editor, *Proc.* 9^{th} *Intl. Conference on Computer Aided Verification (CAV'97)*, Lect. Notes in Comp. Sci., pages 268 – 278. Springer-Verlag, 1997.

[HT96] M. R. Henzinger and J. A. Telle. Faster algorithms for the nonemptiness of street automata and for communication protocol prunning. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory*, pages 10–20, 1996.

[Lic91] O. Lichtenstein. *Decidability, Completeness, and Extensions of Linear Time Temporal Logic*. PhD thesis, Weizmann Institute of Science, 1991.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proc. 8th Int. Colloq. Aut. Lang. Prog.*, volume 115 of *Lect. Notes in Comp. Sci.*, pages 264–277. Springer-Verlag, 1981.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. Logics of Programs*, volume 193 of *Lect. Notes in Comp. Sci.*, pages 196–218. Springer-Verlag, 1985.

[MP91a] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. Henzinger, editors, *Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci., pages 184–195. Springer-Verlag, 1996.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in *cesar*. In M. Dezani-Ciancaglini and M. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lect. Notes in Comp. Sci.*, pages 337–351. Springer-Verlag, 1982.

[SdRG89] F.A. Stomp, W.-P. de Roever, and R.T. Gerth. The μ-calculus as an assertion language for fairness arguments. *Inf. and Comp.*, 82:278–322, 1989.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, 1986.