Formal Methods in System Design, 28, 57–84, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Model Checking with Strong Fairness*

YONIT KESTEN† ykesten@bgumail.bgu.ac.il
Ben Gurion University

AMIR PNUELI amir.pnueli@weizmann.ac.il
LI-ON RAVIV
ELAD SHAHAR elad.shahar@weizmann.ac.il
Weizmann Institute of Science

Abstract. In this paper we present a coherent framework for symbolic model checking of linear-time temporal logic (LTL) properties over finite state reactive systems, taking full fairness constraints into consideration. We use the computational model of a *fair discrete system* (FDS) 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 FDS 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 μ-calculus or CTL. On the technical level, it extends previous methods by dealing with compassion at the algorithmic level instead of either adding it to the specification, or transforming compassion to justice.

Finally, we extend CTL* with past operators, and show that the basic symbolic feasibility algorithm presented here, can be used to model check an arbitrary CTL* formula over an FDS with full fairness constraints.

Keywords: model checking, temporal logic, LTL, CTL, fairness, fair discrete systems, temporal testers

1. Introduction

Two kinds of temporal logics have been proposed over the years for specifying the properties of reactive systems: the *linear time* logic LTL [8] and the *branching time* variant CTL [2]. 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 [2, 23].

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 discrete 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

*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.
†Author to whom all correspondence should be addressed.

several copies. This is the reason why the development 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 [2], and were developed for the branching-time temporal logic CTL. The LTL version of these algorithms was developed in [16] for the future fragment of propositional LTL (PTL), and extended in [18] to the full PTL. The basic fixed-point computation algorithm for the identification of fair computations presented in [16], was developed independently in [6] 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 Büchi to a Streett automaton, we can view the algorithms presented in [6] and [16] as algorithms for checking the emptiness of Streett automata [26]. An improved algorithm solving the related problem of emptiness of Streett automata, was later presented in [10]. The development of the impressively efficient symbolic verification methods and their application to CTL [1] raised the question whether a similar approach can be applied to PTL. The first satisfactory answer to this question is given in [1], by showing a reduction of PTL (future fragment) model checking into μ-calculus model checking. A similar transformation from LTL to CTL model checking, is presented in [3]. The advantage 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 symbolic 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., [7, 8, 17]). 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 [25]. The only question remaining is how practical this is.

In this paper we present an approach to the symbolic model checking of LTL formulas, which takes into account full fairness, including both justice and compassion. The approach is self-contained and does not depend on a reduction to either μ-calculus or CTL model checking (as in [1] and [3], respectively). The main advantage of such a self-contained approach is that the end users no longer need to deal with two different kinds of logics.

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 [1, 3] which, in turn, is an extension of the *statification* method used in [19] and [21] to deal with the past fragment of LTL. The model-checking problem is then reduced into the question of feasibility of an FDS. The symbolic feasibility algorithm, similar to the enumerative algorithm of [16], identifies all computations satisfying a given set of fairness constraints. This involves the identification of all fair strongly connected subgraphs (SCS). However, while the enumerative algorithm identifies each SCS separately, the BDD-based symbolic algorithm is more efficient, identifying all states participating in some fair SCS simultaneously. Our symbolic algorithm can be viewed as a straightforward implementation of the nested fixed-point characterization of Emerson-Lei for fully fair computations [5], as opposed to the CTL model checkers which consider only the weak-fairness part of this characterization.

Other works related to the approach developed here are presented in [9, 11], where BDD-based symbolic algorithms for bad cycle detection are presented. They are based on the fixed-point characterization of Emerson-Lei. These algorithms solve the problem of finding all those cycles within the computation graph, which satisfy a given set of weak fairness constraints. [9] gives heuristics which improve the performance of the Emerson-Lei algorithm. We use the same heuristics in our algorithm, while dealing with both types of fairness constraints. [9, 11] can deal with strong fairness by reduction to weak fairness. According to the automata-theoretic view, [9] presents a symbolic algorithm for the problem of emptiness of Büchi automata, while the algorithms presented here provide a symbolic solution to the emptiness problem of Streett automata.

An algorithm for dealing with compassion at the algorithmic level is presented in [14], however it is enumerative whereas our work provides a symbolic solution.

In [6], Emerson and Lei observed that the problem of CTL* model checking of finite state systems can be resolved by recursive calls to an LTL model-checking algorithm. Taking a similar approach, we augment CTL* with past operators and show that the symbolic feasibility algorithm presented here, can be used to model check an arbitrary CTL* formula over a finite state FDS $\mathcal{D}$, taking the full fairness constraints of $\mathcal{D}$ into consideration.

We present a symbolic algorithm for finding a counter example for an LTL formula. The counter example is a finite path from an initial state to a fair cycle. In [4] a similar algorithm is presented in the context of CTL, evaluating either a witness to an existential property or a counter example for a universal property. The main difference between our algorithm and [4] is that we search for a fair cycle embedded within a terminal strongly connected component while [4] look for a cycle embedded within the entire set of feasible states. Their search may fail several times before a fair cycle is found. Our search is guaranteed to succeed on the first trial.

The rest of the paper is organized as follows. In Section 2 we present the computational model of *fair discrete systems* (FDS). In Section 3 we present PTL, the propositional fragment of linear temporal logic, including the past operators. Next, in Section 4 we discuss the construction of a *tester* for a PTL formula φ, which is an FDS characterizing all the sequences which satisfy φ. Having transformed the model-checking problem into the feasibility problem of an FDS, we present the symbolic feasibility algorithm in Section 5, followed by an algorithm for extracting a witness (a counter example) in Section 6. In Section 7 we augment CTL* with past operators and use the feasibility algorithm to model check an arbitrary CTL* formulas over a finite state FDS. We conclude in Section 8 with some experimental results, comparing the different methods used to deal with compassion requirements.

A (partial) conference version of this paper appeared in [13].

2. Fair discrete systems

As a computational model for reactive systems, we take the model of *fair discrete system* (FDS). The computational model is used for modeling both the verified system and the temporal properties. The FDS model replaces the earlier model of *fair transition system* (FTS) presented in [20] and [21]. The main difference between these two models is in the representation of fairness constraints. The advantage of the new representation is that it enables a unified representation of fairness constraints arising from both the system being verified, and the temporal property.

An FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ consists of the following components:

- $V = \{u_1, \ldots, u_n\}$: A finite set of typed *state variables* ranging 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 $\mathcal{D}$. A state is called *initial* if it satisfies Θ.
- ρ: A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to its $\mathcal{D}$-successor $s' \in \Sigma$ by referring to both unprimed and primed versions of the state variables. The transition relation $\rho(V, V')$ identifies state s' as a $\mathcal{D}$-*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 x' as $s'[x]$.
- $\mathcal{J} = \{J_1, \ldots, J_k\}$: A set of assertions expressing the *justice* (*weak fairness*) requirements. The justice requirement $J \in \mathcal{J}$ stipulates that every computation contains infinitely many J-states (states satisfying J).
- $\mathcal{C} = \{\langle p_1, q_1\rangle, \ldots \langle p_n, q_n\rangle\}$: A set of assertions expressing the *compassion* (*strong fairness*) requirements. The compassion requirement $\langle p, q\rangle \in \mathcal{C}$ stipulates that every computation containing infinitely many p-states also contains infinitely many q-states.

Let σ: $s_0, s_1, \ldots,$ be a sequence of states, φ be an assertion, and $j \geq 0$ be a natural number. We say that j is a φ-position of σ if s_j is a φ-state, namely, if φ holds on s. Let $\mathcal{D}$ be an FDS for which the above components have been identified. We define a *run* of $\mathcal{D}$ to be an infinite sequence of states $\sigma : s_0, s_1, \ldots,$ satisfying the requirements:

- *Initiality*: s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution*: For each $j = 0, 1, \ldots,$ the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.

A run of $\mathcal{D}$ is called a *computation* if it satisfies the following fairness requirements:

- *Justice*: For each $J \in \mathcal{J}$, σ contains infinitely many J-positions
- *Compassion*: For each $\langle p, q\rangle \in \mathcal{C}$, if σ contains infinitely many p-positions, it also contain infinitely many q-positions.

We denote by *Comp* $(\mathcal{D})$ the set of all computations of $\mathcal{D}$.

A state s is said to be $\mathcal{D}$-*reachable* if it participates in a run of $\mathcal{D}$. We say that a state is $\mathcal{D}$-*feasible* if it participates in some computation of $\mathcal{D}$. An FDS $\mathcal{D}$ is *feasible* if $\mathcal{D}$ has at least one computation. We say that an FDS $\mathcal{D}$ is *viable* if every $\mathcal{D}$-reachable state is $\mathcal{D}$-feasible. Note that the FDS model does not guarantee viability.

Parallel Composition of FDS'S

Fair discrete systems can be composed in parallel. Let $\mathcal{D}_i = \langle V_i, \Theta_i, \rho_i, \mathcal{J}_i, \mathcal{C}_i \rangle, i \in \{1, 2\}$, be two fair discrete systems. Two versions of parallel composition are used.

Asynchronous composition is used to assemble an asynchronous system from its components. We define the asynchronous parallel composition of two FDS'S to be

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

where,

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

We can view the execution of $\mathcal{D}$ as the interleaved execution of $\mathcal{D}_1$ and $\mathcal{D}_2$.

Synchronous composition is used in some cases, to assemble a system from its components (in particular when considering hardware designs which are naturally synchronous). However, our primary use of synchronous composition is for combining a system with a *tester* T_φ for an LTL formula φ (see Section 4). We define the synchronous parallel composition of two FDS'S to be

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

where,

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

The synchronous parallel composition of systems $\mathcal{D}_1$ and $\mathcal{D}_2$ is a new system $\mathcal{D}$, each of whose basic actions consists of the joint execution of an action of $\mathcal{D}_1$ and an action of $\mathcal{D}_2$. We can view the execution of $\mathcal{D}$ as the *joint execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$.

3. Linear temporal logic

As a requirement specification language for reactive systems we take the propositional fragment of *linear temporal logic* (PTL) [20].

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:

$$\begin{array}{ll} \bigcirc - \text{Next} & \ominus - \text{Previous} \\ \mathcal{U} - \text{Until} & \mathcal{S} - \text{Since} \end{array}$$

We refer to the set of variables that occur in a formula $\mathcal{P}$ as the *vocabulary of p*. A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state s_j provides an interpretation for the vocabulary of 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,
  $(\sigma, j) \models p \Leftrightarrow s_j \models p$
  That is, we evaluate p locally, using the interpretation given by s_j.
- $(\sigma, j) \models \neg p \Leftrightarrow (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \Leftrightarrow (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \Leftrightarrow (\sigma, j+1) \models p$
- $(\sigma, j) \models p\mathcal{U}q \Leftrightarrow$ for some $k \geq j, (\sigma, k) \models q$,
  and for every i such that $j \leq i < k, (\sigma, i) \models p$
- $(\sigma, j) \models \ominus p \Leftrightarrow j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p\mathcal{S}q \Leftrightarrow$ for some $k \leq j, (\sigma, k) \models q$,
  and for every i such that $k < i \leq j, (\sigma, i) \models p$

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.

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

Additional temporal operators may be defined as follows:

$$\begin{aligned} \Diamond p &= \mathrm{T}\mathcal{U}p - \text{Eventually}\, p \\ \Box p &= \neg\Diamond\neg p - \text{Always, henceforth } p \\ p\mathcal{W}q &= \neg((\neg q)\mathcal{U}(\neg p \wedge \neg q)) - \text{Waiting-for, unless, weak until} \end{aligned}$$

4. Construction of testers for LTL formulas

In this section, we present the construction of a *tester* for a LTL formula φ, which is an FDS T_φ characterizing all the sequences which satisfy φ. Without loss of generality, assume that the only temporal operators occurring in φ are $\bigcirc, \mathcal{U}, \ominus$ and $\mathcal{S}$.

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 FDS T_φ is given by

$$T_\varphi : \langle V_\varphi, \Theta_\varphi, \rho_\varphi, \mathcal{J}_\varphi, \mathcal{C}_\varphi \rangle,$$

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

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

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

$$\Theta_\varphi = \textit{past-init}(\varphi),$$

where
$$\textit{past-init}(\varphi) = \bigwedge_{\ominus p \in \varphi} \neg x_{\ominus p} \wedge \bigwedge_{p\mathcal{S}q \in \varphi} \left(x_{p\mathcal{S}q} \leftrightarrow \chi(q)\right)$$

Thus, the initial condition requires 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, *past-init*(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 satisfying q.

Note that, unlike the definition of testers presented in [12, 13], the assertion $\chi(\varphi)$ is not a conjunct of Θ_φ. Namely, the initial condition of a tester T_φ does not assert $\chi(\varphi)$. This will permit the use of algorithm FEASIBLE presented in Section 5, for model checking both LTL and CTL* properties, as discussed in Section 7.

Transition relation

The transition relation of T_φ is given by

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

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 $\chi(\psi)$. The conjuncts of the transition relation corresponding to the Since and the *Until* operators are based on the following expansion formulas:

$$p\mathcal{S}q \Leftrightarrow q \vee (p \wedge \ominus(p\mathcal{S}q)) \quad p\mathcal{U}q \Leftrightarrow q \vee (p \wedge \bigcirc(p\mathcal{U}q))$$

where $\Leftrightarrow$ denotes *congruence*, namely $(a \Leftrightarrow b) = \square(a \leftrightarrow b)$.

Fairness requirements

For each formula $p\mathcal{U}q \in \varphi$, we include in $\mathcal{J}$ the disjunction

$$\chi(q) \vee \neg x_{p\mathcal{U}q}$$

This justice requirement 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 $\tilde{\sigma}$ is a *U-variant* of sequence σ if σ and $\tilde{\sigma}$ 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 $\sigma = s_0, \ldots$ satisfies the temporal formula φ iff σ is an X_φ-variant of a computation $\tilde{\sigma} = \tilde{s}_0, \ldots$ of T_φ, such that $\tilde{s}_0 \models \chi(\varphi)$.*

5. Checking for feasibility

In this section we present a symbolic algorithm for computing the set of $\mathcal{D}$-feasible states. The symbolic algorithm presented here is inspired by the full state-enumeration algorithm originally presented in [16] and [6] (for full explanations and proofs see [15] and [21]). The enumerative algorithm was designed for LTL model checking, and was concerned with checking feasibility of an FDS. Since we want to use the same basic algorithm for both LTL and CTL* model checking, our basic symbolic algorithm computes the set of $\mathcal{D}$-feasible states, from which the feasibility of $\mathcal{D}$ is trivially obtained. The enumerative algorithm constructs a *state-transition graph* $G_\mathcal{D}$ for $\mathcal{D}$. This is a directed graph whose nodes are all the $\mathcal{D}$-reachable states, and whose edges connect node s to node s' iff s' is a $\mathcal{D}$-successor of s. If system $\mathcal{D}$ has a computation it corresponds to an infinite path in the graph $G_\mathcal{D}$ which starts at a $\mathcal{D}$-initial state. We refer to such paths as *initialized paths*.

Subgraphs of $G_\mathcal{D}$ can be specified by identifying a subset $S \subseteq G_\mathcal{D}$ of the nodes of $G_\mathcal{D}$. It is implied that as the edges of the subgraph we take all the original $G_\mathcal{D}$-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 (scs) which is both just and compassionate.

For π, an infinite initialized path in $G_\mathcal{D}$, we denote by *Inf*(π) the set of states which appear infinitely many times in π. The following claim, which is proved in [15], connects computations of $\mathcal{D}$ with fair subgraphs of $G_\mathcal{D}$.

Claim 2. *The infinite initialized path π is a computation of $\mathcal{D}$ iff Inf(π) is a fair subgraph of $G_\mathcal{D}$.*

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 FDS $\mathcal{D}$. 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:

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

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

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

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

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

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

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

Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$, be an FDS. For an assertion φ over V, the system variables of $\mathcal{D}$, we denote by $||\varphi||$ the predicate consisting of all states satisfying φ. Similarly, for an assertion ρ over (V, V'), we denote by $||\rho||$ the relation consisting of all state pairs $\langle s, s'\rangle$ satisfying ρ.

Algorithm FEASIBLE presented in figure 1, denotes by R the transition relation implied by ρ. The predicate variable *new* represents a set of states. We use the shorthand notation

$$R \cap new \overset{\text{def}}{=} R \cap (new \times \Sigma).$$

Algorithm FEASIBLE 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

```
Algorithm FEASIBLE ($\mathcal{D}$) : predicate — Compute the set of $\mathcal{D}$-feasible states
         $new, old$ : predicate
         $R$        : relation
  1.  $old := \emptyset$
  2.  $new := \|\Theta\| \circ \|\rho\|^*$
  3.  $R := \|\rho\| \cap new$
  4.  while ($new \neq old$) do
      begin
  5.     $old := new$
  6.     while ($new \neq new \cap R \circ new$) do
  7.        $new := new \cap (R \circ new)$
  8.     for each $J \in \mathcal{J}$ do
  9.        $new := (R \cap new)^* \circ (new \cap \|J\|)$
 10.     for each $(p, q) \in \mathcal{C}$ do
 11.        $new := (new - \|p\|) \cup (R \cap new)^* \circ (new \cap \|q\|)$
      end
 12.  $new := R^* \circ new$
 13.  return($new$)
```

Figure 1. Algorithm FEASIBLE.

main loop we compute in *new* the universal set of all reachable states in $\mathcal{D}$ and place in R the transition relation implied by ρ, restricted to pairs (s_1, s_2) where s_1 is a reachable state in $\mathcal{D}$.

The main loop (lines 4–11), contains three inner loops. The inner loop at lines 6–7, successively removes from *new* all states which do not have a successor in *new*. The loop at lines 8–9 removes from *new* all states which are not R^*-predecessors of some J-state, for all justice requirements $J \in \mathcal{J}$. The term $R \cap new$ restricts R to pairs (s_1, s_2) where s_1 is currently in *new*. This is done to avoid regeneration of states which have already been eliminated. The loop at lines 10–11, removes from *new* all p-states which are not R^*-predecessors of some q-state for some $(p, q) \in \mathcal{C}$. The term $R \cap new$ restricts R again to pairs (s_1, s_2) where s_1 is currently in *new*.

Finally, line 12 augments *new* with all its R^*-predecessors.

Correctness of the FEASIBLE Algorithm

The following sequence of claims establishes the correctness of the algorithm.

Claim 3 (Termination). *Algorithm* FEASIBLE *terminates.*

Proof: Let us denote by new_i^4 the value of variable *new* on the i'th visit ($i = 1, 2, \ldots$) to line 4 of the algorithm. Since all assignments to variable *new* within the main loop, are of the form new $:= E$ where $E \subseteq new$, it is not difficult to see that $new_2^4 \subseteq new_1^4$. From this, it can be established by induction on i that $new_{i+1}^4, \subseteq new_i^4$, for every $i = 1, 2 \ldots$. It follows that the sequence $|new_1^4| \geq |new_2^4| \geq |new_3^4| \ldots$, 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. □

Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS, s be a $\mathcal{D}$-reachable state and U be the set of states resulting from the application of algorithm FEASIBLE to $\mathcal{D}$.

Claim 4 (Completeness). *If s is $\mathcal{D}$-feasible then $s \in U$.*

Proof: Assume that s is $\mathcal{D}$-feasible. Then by definition, s is on an initialized fair path π in $\mathcal{D}$, From Claim 2, $Inf(\pi)$ is a fair subgraph $S \subseteq G_{\mathcal{D}}$. Namely, S is a non-singular strongly-connected subgraph (scs) 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. Let

$$\tilde{S} : S \cup \{s' | s' \text{ is on an initialized path to a state in } S\}$$

Obviously $s \in \tilde{S}$. Following the operations 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 11, cannot remove any state of S. Consequently, S must remain throughout the process and will be contained in U. Finally, when line 12 is executed, the set *new* is augmented from S to $\tilde{S}$, implying that $s \in U$. □

Claim 5 (Soundness). *If $s \in U$ then s is $\mathcal{D}$-feasible.*

Proof: Assume that $s \in U$. Let $S \subseteq U$ be the set of states in U, reachable from s by a U-path. Since $s \in U$, s is $\mathcal{D}$-reachable. For every $J \in \mathcal{J}$, s can reach a J-state by a path fully contained within S. For every $(p, q) \in \mathcal{C}$, either s is not a p-state, or s can reach a q-state by an S-path.

Let us decompose S into maximal strongly-connected subgraphs (SCS). At least one subgraph S_t is *terminal* in this decomposition, in the sense that every S-edge exiting an S_t-state also leads to an S_t-state. We argue that S_t is fair. By definition, it is strongly connected. It cannot be singular, because it would consist of a single state that would have been removed on the last execution of the loop at lines 6–7. Let r be an arbitrary state within S_t. For every $J \in \mathcal{J}$, r can reach some J-state $\tilde{r} \in U$ by an S-path. Since S_t is terminal within S, this path must be fully contained within S_t and, therefore, $\tilde{r} \in S_t$. In a similar way, we can show that S_t satisfies all the compassion requirements. We can conclude that s is on an initialized path to a fair subgraph, which establishes that s is $\mathcal{D}$-feasible. □

Claims 4 and 5 lead to the following conclusion:

Corollary 6. *s is $\mathcal{D}$-feasible iff $s \in U$.*

Corollary 7. *$\mathcal{D}$ is feasible iff $U \cap \|\Theta\| \neq \emptyset$.*

Proof: A direct result of Corollary 6 and the definition of feasibility. □

Algorithm FEASIBLE for CTL* Versus LTL

Algorithm FEASIBLE presented in figure 1 is designed for model-checking both LTL and CTL* properties over an FDS. In [13], we present a similar algorithm designed for model-checking LTL properties only. An algorithm for LTL model checking, needs to assert the existence (non-existence) of a reachable fair SCS in some FDS $\mathcal{D}$. The original enumerative algorithms of [6] and [16] were based on recursive exploration of strongly connected subgraphs, ensuring closure under both successors and predecessors. As the work in [13] shows, it is possible to relax the requirement of bi-directional closure into either closure under predecessors and search for *terminal* SCS components, or, alternatively, closure under successors and search for *initial* SCS components. This idea, which may be worth exploring even in the enumerative case, has been explored extensively in the context of symbolic model checking (see [24]).

In this paper however, algorithm FEASIBLE is designed for model-checking both LTL and CTL* properties, as discussed in Section 7. In the case of CTL*, we need to evaluate the set of feasible states in $\mathcal{D}$, namely the set of all states on some reachable fair SCS, or on a finite initialized path to a reachable fair SCS. In this case, we can no longer choose arbitrarily between forward and backward closure evaluation. The only appropriate choice is backward closure, which guarantees that all terminal SCS are fair. This is the version of algorithm FEASIBLE presented in figure 1.

Model Checking LTL Properties

Using algorithm FEASIBLE, we can now model check an LTL property φ over an FDS $\mathcal{D}$ as follows.

- Construct the tester $T_{\neg\varphi}$.
- Construct the synchronous parallel composition $\mathcal{A} : \mathcal{D}|||T_{\neg\varphi}$. Let $\Theta_{\mathcal{A}}$ be the initial condition of the FDS $\mathcal{A}$.
- Evaluate FEASIBLE $(\mathcal{D}|||T_{\neg\varphi}) \cap ||\Theta_{\mathcal{A}} \wedge \chi(\neg\varphi)||$.

The verification is based on the following Claim.

Claim 8 $\mathcal{D} \models \varphi$ *iff* FEASIBLE $(\mathcal{D}|||T_{\neg\varphi}) \cap ||\Theta_{\mathcal{A}} \wedge \chi(\neg\varphi)|| = \emptyset$

The proof of equivalent claims can be found in [16, 26].

6. 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 or a counter-example.* Since we reduced the problem of checking $\mathcal{D} \models \varphi$ to checking the feasibility of $\mathcal{D}|||T_{\neg\varphi}$, such a witness can be provided by a computation of the combined FDS $\mathcal{D}|||T_{\neg\varphi}$.

In the following we present an algorithm which produces a computation of an FDS 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 $L_2 = (s_b, \ldots, s_c)$, we denote by $L_1 * L_2$ their *concatenation*, defined by $L_1 * L_2 = (s_1, \ldots, s_a, s_b, \ldots, s_c)$. 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 figure 2, 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 3 we present an algorithm which produces a computation of a given FDS. Although a computation is an infinite sequence of states, if $\mathcal{D}$ is feasible, it always has an *ultimately periodic* computation of the following form:

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

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
  new   : predicate
  bp[]  : array of predicates
  pos   : integer
  s     : state
  new := destination
  pos := 1;   bp [pos] := new
  while (bp [pos] $\cap$ source $= \emptyset$) do
  begin
     bp [pos + 1] := bp [pos] $\cup$ $R \circ$ bp [pos]
     pos := pos + 1
  end
  s := choose (bp [pos] $\cap$ source)
  L := (s)
  while (pos > 1) do
  begin
     pos := pos − 1
     s := choose (s $\circ R \cap$ bp [pos])
     L := L $*$ (s)
  end
  return L
```

Figure 2. Function *path.*

```
Algorithm WITNESS ($\mathcal{D}$) : [list, list]  —  Extract a witness for a feasible FDS.
        final            : predicate
        R                : relation
        prefix, period   : list
        s                : state
  1.  final := FEASIBLE ($\mathcal{D}$)
  2.  if (final $= \emptyset$) then return $(\Lambda, \Lambda)$
  3.  $R := \|\rho\| \cap$ (final $\times$ final)
  4.  s := choose(final)
  5.  while ($\{s\} \circ R^* - R^* \circ \{s\} \neq \emptyset$) do
  6.     s := choose($\{s\} \circ R^* - R^* \circ \{s\}$)
  7.  final := $\{s\} \circ R^*$
  8.  $R := R \cap$ (final $\times$ final)
  9.  prefix := path($\|\Theta\|$, final, $\|\rho\|$)
 10.  period := (choose({last(prefix)} $\circ R$))
 11.  for each $J \in \mathcal{J}$ do
 12.     if (list-to-set(period) $\cap \|J\| = \emptyset$) then
 13.        period := period $*$ path({last(period)}, final $\cap \|J\|$, R)
 14.  for each $(p, q) \in \mathcal{C}$ do
 15.     if (list-to-set(period) $\cap \|q\| = \emptyset \ \wedge$ final $\cap \|p\| \neq \emptyset$) then
 16.        period := period $*$ path({last(period)}, final $\cap \|q\|$, R)
 17.  period := period $*$ path({last(period)}, {last(prefix)}, R)
 18.  return (prefix, period)
```

Figure 3. Algorithm WITNESS.

The algorithm starts by checking whether FDS $\mathcal{D}$ is feasible. It uses Algorithm FEASIBLE to perform this check. If $\mathcal{D}$ is found to be infeasible, the algorithm exits while providing a pair of empty lists as a result.

If $\mathcal{D}$ is found to be feasible, we store in *final* the set of states returned by FEASIBLE. This set contains the set of all states participating in a computation of $\mathcal{D}$. We restrict the transition relation R to pairs (s_1, s_2) where both s_1 and s_2 are states within *final*. Next, we perform a search for a *terminal maximal strongly connected subgraph* (terminal 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 $\{s\} \circ R^* \subseteq R^* \circ \{s\}$. i.e. a state all of whose R^*-successors are also R^*-predecessors. This is done by successively replacing s by a state $s \in \{s\} \circ R^* - R^* \circ \{s\}$, as long as the set of s-successors is not contained in the set of s-predecessors. Eventually, execution of the loop must terminate when s reaches a terminal MSCS within *final*. Termination is guaranteed because each such replacement moves the state from one MSCS to a proceeding MSCS in the canonical decomposition of *final* into MSCS's.

A central point in the proof of correctness of Algorithm FEASIBLE established that any terminal 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 R 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 the last state of the prefix, *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 states, 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 the MSCS *final* contains some p-state. Finally, in line 17, we complete the path to form a closed cycle by looping back to *last*(*prefix*).

7. Symbolic model checking CTL* properties

In the following, we show that algorithm FEASIBLE can be used to model check an arbitrary CTL* formula over a finite state FDS, taking weak and strong fairness constraints into consideration. We define CTL* with both future and past temporal operators. We denote the fragment of CTL* without the past operators as the *future fragment* of CTL*.

An enumerative algorithm for model checking the future fragment of CTL* is presented in [6]. In this work, Emerson and Lei show that model checking a CTL* formula over a finite state system, can be performed by recursive calls to an LTL model checker. We take a similar approach, using algorithm FEASIBLE to verify an arbitrary CTL* formula over a finite state FDS $\mathcal{D}$.

In the following, we first present the syntax and semantics of the logic, then discuss the use of algorithm FEASIBLE for verifying CTL* properties.

7.1. The Logic CTL*

A propositional CTL* formula is constructed out of propositions to which we apply the boolean operators, temporal operators and path quantifiers. The temporal operators are the same operators presented in Section 3 for LTL. The *path quantifiers* are E_f and A_f, as defined below.

A CTL* formula p is interpreted over the state graph (Kripke structure) generated by an FDS $\mathcal{D}$. In the following, we use the term *path* in $\mathcal{D}$ as synonymous to a *computation* of $\mathcal{D}$.

There are two types of formulas in CTL*: *State formulas* which are interpreted over states and *path formulas* which are interpreted over paths. Let $\mathcal{P}$ be a finite set of propositions. The syntax of a CTL* formula is defined inductively as follows.

State formulas:

- Every proposition $p \in \mathcal{P}$ is a state formula.
- If p is a *path formula*, then $E_f p$ and $A_f p$ are state formulas.
- If p and q are state formulas then so are $\neg p$ and $p \vee q$.

Path formulas:

- Every state formula is a path formula.
- If p and q are path formulas then so are $\neg p$, $p \vee q$, $\bigcirc p$, $p\mathcal{U}q$, $\ominus p$ and $p\mathcal{S}q$.

The formulas of CTL* are all the state formulas generated by the above rules.

A state formula of the form Qp, where Q is a path quantifier and p is a path formula containing no path quantifiers is called a *basic state formula.* A basic state formula of the form $A_f\psi(E_f\psi)$ is called a basic *universal* (*existential*) formula. Note that the set of basic universal formulas corresponds to the set of linear temporal logic formulas (LTL).

The semantics of a CTL* formula is defined inductively as follows. State formulas are interpreted over states in $\mathcal{D}$. We define the notion of a path formula p holding at a state s in $\mathcal{D}$, denoted $(\mathcal{D}, s) \models p$, as follows:

- For an assertion p
- $(\mathcal{D}, s) \models p, \Leftrightarrow s \models p$
- $(\mathcal{D}, s) \models \neg p \Leftrightarrow (\mathcal{D}, s) \not\models p$
- $(\mathcal{D}, s) \models p \vee q \Leftrightarrow (\mathcal{D}, s) \models p$ or $(\mathcal{D}, s) \models q$
- $(\mathcal{D}, s) \models E_f p \Leftrightarrow (\mathcal{D}, \pi, j) \models p$ for some path $\pi = \pi_0, \pi_1 \ldots \in Comp(\mathcal{D})$, and position $j \geq 0$ satisfying $\pi_j = s$.

The universal path quantifier is defined by $A_f p = \neg E_f \neg p$. Note that we have only defined the fair versions A_f and E_f of the path quantifiers. If one wants to verify a formula $A\varphi$ over an FDS $\mathcal{D}$, it can be done by verifying $A_f\varphi$ over $\mathcal{D}_{\text{unfair}}$, which is obtained from $\mathcal{D}$ by removing all fairness requirements.

Path formulas are interpreted over a path (computation) in $\mathcal{D}$. We define the notion of a path formula p holding at position $j \geq 0$ of a path π in $Comp(\mathcal{D})$, denoted $(\mathcal{D}, \pi, j) \models p$, as follows:

- For a state formula p,
  $(\mathcal{D}, \pi, j) \models p \Leftrightarrow (\mathcal{D}, s) \models p$, for $s = \pi_j$
- $(\mathcal{D}, \pi, j) \models \neg p \Leftrightarrow (\mathcal{D}, \pi, j) \not\models p$
- $(\mathcal{D}, \pi, j) \models p \vee q \Leftrightarrow (\mathcal{D}, \pi, j) \models p$ or $(\mathcal{D}, \pi, j) \models q$
- $(\mathcal{D}, \pi, j) \models \bigcirc p \Leftrightarrow (\mathcal{D}, \pi, j+1) \models p$
- $(\mathcal{D}, \pi, j) \models p\mathcal{U}q \Leftrightarrow (\mathcal{D}, \pi, k) \models q$ for some $k \geq j$ and $(\mathcal{D}, \pi, i) \models p$ for every $i, j \leq i < k$
- $(\mathcal{D}, \pi, j) \models \ominus p \Leftrightarrow j > 0$ and $(\mathcal{D}, \pi, j-1) \models p$
- $(\mathcal{D}, \pi, j) \models p\mathcal{S}q \Leftrightarrow (\mathcal{D}, \pi, k) \models q$ for some $k, 0 \leq k \leq j$ and $(\mathcal{D}, \pi, i) \models p$ for every $i, k < i \leq j$

Let p be a CTL* formula. We say that p *holds* on $\mathcal{D}$ (p is $\mathcal{D}$-valid), denoted $\mathcal{D} \models p$, if $(\mathcal{D}, s) \models p$, for every initial state s in $\mathcal{D}$. A CTL* formula p is called *satisfiable* if it holds on some model $\mathcal{D}$. A CTL* formula is called *valid* if it holds on all models. Let p and q be CTL* formulas. We use the notations $p \Leftrightarrow q$ and $p \Leftrightarrow q$ as a shorthand for $A_f \square (p \rightarrow q)$ and $A_f \square (p \leftrightarrow q)$ respectively.

7.2. *Model checking basic state formulas*

In the following we show how to model check basic state formulas over an FDS $\mathcal{D}$.

Algorithm FEASIBLE is formulated in set-theoretic terms. However, its implementation obviously represents any set of states by a BDD which also represents an assertion. In the following we use notations such as $\|\alpha\| = $ FEASIBLE $(\mathcal{D})$ in order to refer to the assertion α characterizing the set of states returned by the algorithm.

In figure 4 we present algorithm SAT-E$_f$ which evaluates the set of $\mathcal{D}$-reachable states satisfying a basic existential formula $E_f\varphi$. Note that, since φ is a path formula with no embedded path quantifiers, it is also an LTL formula, for which a tester can be constructed. The following claim states that the set $\|\alpha_{E\varphi}\|$ evaluated by algorithm SAT-E$_f$ is exactly the set of $\mathcal{D}$-reachable states satisfying the CTL* formula $E_f\varphi$.

Calim 9. $(\mathcal{D}, s) \models E_f\varphi$ *iff* $s \in \left\|\alpha_{E\varphi}\right\|$

```
Algorithm SAT-E_f (D, φ) : predicate —
                    — — Compute the set of D-reachable states satisfying E_fφ

1.  Construct the temporal tester $T_\varphi$ for $\varphi$.
2.  Construct the synchronous parallel composition $\mathcal{D}|||T_\varphi$.
3.  Evaluate $\|\psi\| = \|\chi(\varphi)\| \cap$ FEASIBLE($\mathcal{D}|||T_\varphi$)
4.  Project away the auxiliary variables $X_\varphi$ of $T_\varphi$, returning $\|\alpha_{E\varphi}\| = \|\exists X_\varphi : \psi\|$.
```

Figure 4. Algorithm SAT-E$_f$

```
Algorithm SAT-A_f $(\mathcal{D}, \varphi)$ : predicate —
        — — Compute the set of $\mathcal{D}$-reachable states, satisfying $A_f\varphi$

1. Construct the temporal tester $T_{\neg\varphi}$.
2. Construct the synchronous parallel composition $\mathcal{D} \||| T_{\neg\varphi}$.
3. Evaluate $\|\psi\| = \|\chi(\neg\varphi)\| \cap$ FEASIBLE$(\mathcal{D} \||| T_{\neg\varphi})$
4. Project away the auxiliary variables of $T_{\neg\varphi}$ and complement the result,
   returning $\|\alpha_{A\varphi}\| = \|\neg\exists X_{\neg\varphi} : \psi\|$.
```

Figure 5. Algorithm SAT-A$_f$.

Corollary 10. $\mathcal{D} \models \quad \alpha_{E\varphi} \Leftrightarrow E_f\varphi$

Let $\mathcal{D}$ be an FDS and $A_f\varphi$ be a basic universal formula. To evaluate the set of $\mathcal{D}$-reachable states satisfying the formula, we use the CTL* congruence $A_f\varphi \Leftrightarrow \neg E_f\neg\varphi$.
In figure 5 we present algorithm SAT-A$_f$ which evaluates the set of $\mathcal{D}$-reachable states satisfying a basic universal formula $A_f\varphi$
The following claim states that the set $\|\alpha_{A\varphi}\|$ evaluated by algorithm SAT-A$_f$ is exactly the set of $\mathcal{D}$-reachable states satisfying the CTL* formula $A_f\varphi$.

Claim 11. $(\mathcal{D}, s) \models A_f\varphi$ *iff* $s \in \|\alpha_{A\varphi}\|$

Corollary 12. $\mathcal{D} \models \quad \alpha_{A\varphi} \Leftrightarrow A_f\varphi$

The two algorithms SAT-E$_f$ and SAT-A$_f$ can be combined into a single algorithm SAT-BASIC ($\mathcal{D}$, φ), defined by

$$\text{SAT-BASIC } (\mathcal{D}, E_f\varphi) = \text{SAT-E}_f(\mathcal{D}, \varphi)$$
$$\text{SAT-BASIC } (\mathcal{D}, A_f\varphi) = \text{SAT-A}_f(\mathcal{D}, \varphi)$$

7.3. Decomposing an arbitrary CTL* *formula into basic formulas*

Consider an arbitrary (non basic) CTL* formula p which we wish to verify over an FDS $\mathcal{D}$. Following [6], we reduce the task of verifying formula p into simpler subtasks, each required to verify a basic state formula over $\mathcal{D}$.

In figure 6 we present algorithm VALID-CTL* for the verification of an arbitrary CTL* formula ψ over an FDS $\mathcal{D}$. The algorithm consists of a while loop, where at each iteration of the loop formula ψ is reduced to a new formula in which all occurrences of some basic state formula φ within ψ, are replaced by an assertion congruent to φ. The loop terminates when ψ is reduced to an assertion $\tilde{\psi}$ that characterizes the set of states in $\mathcal{D}$ satisfying ψ. Formula ψ is $\mathcal{D}$-valid if $\|\Theta_{\mathcal{D}} \subseteq \|\tilde{\psi}\|$ or equivalently $\|\Theta_{\mathcal{D}}\| \rightarrow \tilde{\psi}$. The following claim states the soundness of algorithm VALID-CTL*.

```
Algorithm VALID-CTL* $(\mathcal{D},\psi)$ : Boolean — T if $\psi$ is $\mathcal{D}$-valid, F otherwise.
1. while ($\psi$ not an assertion) do
2.    begin
3.       identify a basic state formula $\varphi$ in $\psi$
4.       evaluate $\|\alpha\|$ := SAT-BASIC $(\mathcal{D},\varphi)$
5.       $\psi := \psi[\varphi \leftarrow \alpha]$
6.    end
7. return($\Theta_{\mathcal{D}} \rightarrow \psi$)
```

Figure 6. Algorithm VALID-CTL*.

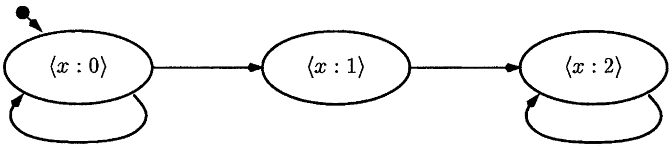

Figure 7. An example system $\mathcal{D}$.

Let $\mathcal{D}$ be an FDS and ψ be an arbitrary CTL* formula, then

Claim 13. $\mathcal{D} \models \psi$ *iff* VALID-CTL* $(\mathcal{D}, \psi)$ *returns* T.

Example. Consider the system $\mathcal{D}$ presented in figure 7. This system has a single state variable x and no fairness requirements. For this system we wish to prove the property $f : E_f \Box E_f \diamond (x = 1)$, claiming the existence of a computation from each of whose states it is possible to reach a state at which $x = 1$.

Using algorithm VALID-CTL*, the task of verifying the non-basic formula

$$E_f \Box E_f \diamond (x = 1)$$

is reduced into the following tasks:

R1. Evaluate $\|\alpha_1\|$ = SAT-BASIC $(\mathcal{D}, E_f \diamond (x = 1))$. This yields $\alpha_1 : (x = 0)$.
R2. Evaluate $\|\alpha_2\|$ = SAT-BASIC $(\mathcal{D}, E_f \Box \alpha_1)$. This yields $\alpha_2 : (x = 0)$.
R3. Evaluate $\Theta_{\mathcal{D}} \rightarrow \alpha_2$. This yields T.

8. Experimental results

The algorithms described in this paper were implemented within the TLV system [22]. In the following section, we summarize our experimental results for algorithm FEASIBLE.

The experiments were carried on a Sun Ultra with 1 Gigabyte of memory. We limit our attention to LTL properties since our intention is to test the performance of algorithm FEASIBLE, which is the same for LTL and CTL*.

8.1. Compassion at the algorithmic level

In order to test whether compassion at the algorithmic level yields better performance, we verify several examples which require compassion, using three different verification methods:

1. Verifying compassion at the algorithmic level. We denote this methods by NT (*no transformation*).
2. Transforming compassion to justice. From an automata theoretic perspective this transforms a Streett automata into a Büchi automata. We denote this method by CJ (*compassion transformed to justice*).
3. Replacing compassion requirements in the system by adding an antecedent to the verified property. We denote this method by CA (*compassion as antecedent*).

In method CJ, every compassion requirement is transformed into a justice requirement, at the price of introducing an extra boolean variable. Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be an FDS. For every $\langle p_i, q_i\rangle \in \mathcal{C}$ we introduce a new variable r_i and modify $\mathcal{D}$ as follows:

$$\begin{array}{ll}
\tilde{V}: & V \cup \{r_i\} \\
\tilde{\Theta}: & \Theta \wedge (r_i = F) \\
\tilde{\rho}: & \rho \wedge (r_i \rightarrow (\neg p_i \wedge r_i')) \\
\tilde{\mathcal{J}}: & \mathcal{J} \cup \{(r_i \vee q_i)\} \\
\tilde{\mathcal{C}}: & \mathcal{C} - \{(p_i, q_i)\}
\end{array}$$

Method CA replaces a set of compassion requirements by a conjunction of LTL formulas, added as an antecedent to the verified property. Let $\mathcal{D}$ be an FDS, φ be a formula we wish to verify over $\mathcal{D}$ and $\mathcal{C}$ be the set of compassion requirements in $\mathcal{D}$. We replace the verification of $\mathcal{D} \models \varphi$ by the following verification task:

$$\mathcal{D}_{(-\mathcal{C})} \models \left(\bigwedge_{\langle p,q\rangle \in C} (\Box\Diamond p \rightarrow \Box\Diamond q) \right) \rightarrow \varphi$$

where $\mathcal{D}_{(-c)}$ is the FDS $\mathcal{D}$ from which all the compassion requirements have been removed.

Note that a similar method can be applied to CTL* formulas. Let ψ be a CTL* property, and c be defined as follows:

$$c = \bigwedge_{\langle p,q\rangle \in C} (\Box\Diamond p \rightarrow \Box\Diamond q)$$

We modify the calls to SAT-E$_F$ and SAT-A$_f$ in algorithm VALID-CTL* as follows:

- For a basic existential formula, call SAT-E$_f(\mathcal{D}_{(-\mathcal{C})}, c \wedge \varphi)$.
- For a basic universal formula, call SAT-A$_f(\mathcal{D}_{(-\mathcal{C})}, c \rightarrow \varphi)$.

8.1.1. Feasibility of parameterized programs. The programs we use for experimentation are *parameterized* programs, of the form

$$S(n) : P[1] \| P[2] \| \ldots \| P[n]$$

which are verified for different values of n.

Consider program DINE presented in figure 8. This program is a symmetric solution to the dining philosophers problem, using semaphores for coordination between processes. Program DINE satisfies the safety requirement of mutual exclusion, stating that no neighboring philosophers can dine at the same time. However, this program fails to satisfy the liveness requirement of *accessibility* for the first process, stating that if the philosopher wishes to dine, it will eventually do so, as specified by:

$$\psi_1 : \Box(at\_\ell_2[1] \rightarrow \diamond at\_\ell_4[1])$$

Program DINE is written in SPL (Simple Programming Language). The process of translating an SPL program to an FDS can be done automatically [21]. The translation associates a compassion requirement with every **request** statement. The execution of the statement **request** s reduces the value of the semaphore s by 1. The statement can be executed only if $s > 1$. For example, the compassion requirement associated with ℓ_2 of process 1 is $\langle at\_\ell_2[1] \wedge c[1] = 1, at\_\ell_3[1] \rangle$. This requirement ensures that if statement ℓ_2 is infinitely often enabled, it is infinitely often taken.

The **non-critical** statement has no corresponding justice or compassion requirement. A process may remain indefinitely in such a statement. For all other statements, the translation associates a justice requirement of the form $\neg\, at\_\ell_i$. In program DINE, statements ℓ_0, ℓ_4, ℓ_5 and ℓ_6 each have corresponding justice requirements.

Note that it may be possible to translate an SPL program to an FDS using fewer justice requirements, however, this typically requires a deeper understanding of the program, and thus manual translation.

```
in     n  : integer where $n \geq 2$
local  c  : array [1..n] where $c = 1$

 n
 ||  P[j] ::  [ $\ell_0$ : loop forever do
j=1               [ $\ell_1$ : non-critical
                    $\ell_2$ : request c[j]
                    $\ell_3$ : request $c[j \oplus_n 1]$
                    $\ell_4$ : critical
                    $\ell_5$ : release c[j]
                    $\ell_6$ : release $c[j \oplus_n 1]$ ] ]
```

Figure 8. Program DINE: The dining philosophers.

```
in     n  : integer where $n > 0$
local  y  : integer where $y = 1$

$\Vert_{i=1}^{n}$ P[i] ::  [ $\ell_0$ : loop forever do
                                [ $\ell_1$ : non-critical
                                  $\ell_2$ : request y
                                  $\ell_3$ : critical
                                  $\ell_4$ : release y     ] ]
```

Figure 9. Program MUX-SEM: Mutual exclusion with semaphores.

As a second example we consider the asymmetric version of the dining philosophers, program DINE-CONTR (dining philosophers with one contrary process), where the behavior of one of the philosophers is reversed, i.e. it first lifts the right fork and then the left one. The accessibility property ψ_1 is valid for program DINE-CONTR.

The third example is program MUX-SEM, presented in figure 9. It implements mutual exclusion by semaphores. The following accessibility property is valid for program MUX-SEM:

$$\psi_2 : \Box(at\_\ell_2[1] \rightarrow \diamond at\_\ell_3[1])$$

In the following experiments we verify the accessibility property ψ_1 for program DINE (Table 1) and DINE-CONTR (Table 2), and the accessibility property ψ_2 for program MUX-SEM (Table 3 and figure 10).

In the tables summarizing our results, we use the following notations:

- **n**—The number of processes for which the parameterized program has been tested.
- **Proc.**—The verification method used for the compassion requirements.
- $|\mathcal{J}|$, $|\mathcal{C}|$—The number of justice and compassion requirements in the FDS $\mathcal{D} ||| T_{\neg\varphi}$.
- **Time**—Timing results (in seconds).
- BDD **Peak**—the maximum number of allocated BDD nodes.
- **Op.**—The number of pre-composition (pre-image) operations invoked by algorithm FEASIBLE.

Table 1. Program DINE.

n	Proc.	$\|\mathcal{J}\|$	$\|\mathcal{C}\|$	Time	BDD Peak	Op.	Ext. Iter.
3	NT	14	6	0.49	10016	474	4
	CJ	20	0	5.34	42674	711	5
	CA	37	0	36.79	414467	1793	4
4	NT	18	8	3.54	17146	1007	6
	CJ	26	0	148.02	223395	1079	5
	CA	49	0	3829.80	2769339	2849	4

Table 2. Program DINE-CONTR.

n	Proc.	$\lvert\mathcal{J}\rvert$	$\lvert\mathcal{C}\rvert$	Time	BDD Peak	Op.	Ext. Iter.
3	NT	14	6	0.98	10016	991	10
	CJ	20	0	2.74	36687	382	5
	CA	37	0	19.40	334345	985	4
4	NT	18	8	3.63	21112	1119	9
	CJ	26	0	29.11	135478	693	6
	CA	49	0	1670.29	2066671	1567	4
5	NT	22	10	20.72	38222	1887	11
	CJ	32	0	1262.11	241312	1238	7
	CA	–	–	–	–	–	–
6	NT	26	12	126.32	87723	2888	13
	CJ	–	–	–	–	–	–
	CA	–	–	–	–	–	–

Table 3. Program MUX-SEM.

n	Proc.	$\lvert\mathcal{J}\rvert$	$\lvert\mathcal{C}\rvert$	Time	BDD Peak	Op.	Ext. Iter.
3	NT	11	3	0.09	5007	168	5
	CJ	14	0	0.23	10002	162	5
	CA	22	0	0.67	22805	184	3
4	NT	14	4	0.16	8726	204	5
	CJ	18	0	0.59	14574	198	5
	CA	29	0	4.40	98191	287	3
5	NT	17	5	0.26	10000	240	5
	CJ	22	0	1.08	21202	234	5
	CA	36	0	61.11	336517	405	3
6	NT	20	6	0.41	10051	276	5
	CJ	26	0	1.80	30369	270	5
	CA	43	0	1296.68	1070510	540	3

- **Ext. Iter.**—The number of external while-loop iterations performed (lines 4–11 of algorithm FEASIBLE).

Rows with no data are of experiments which did not terminate within one hour.

The overall numbers of fairness requirements $|\mathcal{J}| + |\mathcal{C}|$ of the NT and CJ methods are equal, but the compassion requirements of method NT have been transformed to justice requirements in method CJ. The additional justice requirements in method CA are due to the tester, which is generated from a more complex property.

In all our experiments on real systems, method NT has the best results and method CA has (by far) the worst results, both in execution time, and memory consumption.

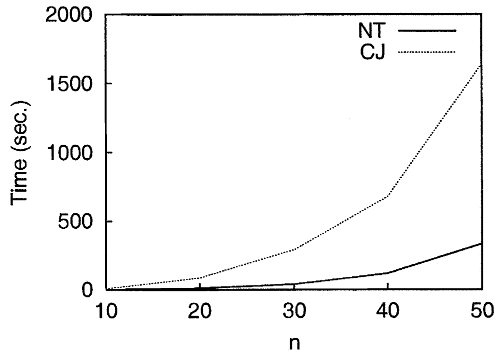

Figure 10. Program MUX-SEM: comparing NT and CJ for larger values of n.

However, in some cases the number of pre-composition operations for method CJ was less than that of method NT.

8.1.2. Feasibility of randomly generated graphs. The experiments for real systems were not conclusive with regard to the comparison between methods NT and CJ. Although method NT has better performance than CJ, the latter sometimes requires significantly fewer pre-composition operations. In this section we present additional experiments for comparing between the two methods.

We can view a system as a digraph where the state space of the system corresponds to the vertex space, and there is a directed edge from u to v in the digraph iff there is a transition from u to v in the system.

The problem in performing experiments over any set of real examples is that they provide a limited range of digraph patterns. Our experiments should not be biased towards any specific design. Therefore, we performed numerous experiments on random digraphs.

On the other hand, using random graphs has disadvantages: this method might be unrealistic since random graphs do not faithfully represent actual programs. That is why we use this method only to complement our experiments on real programs.

We generate random digraphs as suggested in [27]. Given a graph G with vertices V and edges E, the *order* of G is $n = |V|$ and the *density* of G is $d = |E|/|V|$.

In the following tables, $n = 1000$. Each table only changes one parameter, either the size of the set of compassion requirements, or the density of the random graph. Each table entry is the average of 100 experiments.

Table 4 produces results similar to those obtained for real systems. The real systems we checked were parameterized, with compassion requirements associated with each process. Therefore, the number of compassion requirements increases together with the number of processes. In method CJ, each requirement introduces an additional variable, which affects performance of pre-composition operations. Therefore, when the number of pre-composition operations is roughly the same, using method NT is preferable.

Table 4. Varying compassion set size. $d = 1.5$.

	NT			CJ		
$\|\mathcal{C}\|$	Time	Op.	Ext. Iter.	Time	Op.	Ext. Iter.
40	1.33	285.1	2.3	1.14	83.1	2.0
60	1.61	382.7	2.3	1.67	92.6	2.0
80	1.29	301.7	2.2	1.97	79.0	2.0
100	1.89	512.7	2.4	19.46	148.5	2.0
120	2.90	813.1	2.5	105.48	216.9	2.0

Table 5. Varying density. $|\mathcal{C}| = 40$.

	NT			CJ		
Density	Time	Op.	Ext. Iter.	Time	Op.	Ext. Iter.
1.2	0.16	52.9	2.1	0.19	20.1	2.0
1.6	1.92	312.9	2.3	1.32	78.2	2.0
2	8.49	780.3	2.5	4.03	159.0	2.0
2.4	13.25	882.6	2.5	5.90	171.4	2.0
2.8	18.54	1026.9	2.6	7.89	200.5	2.0

In Table 5 method CJ requires less time and less pre-composition operations, in comparison to method NT, as density increases. The reduced number of pre-composition operations compensates for the performance penalty of each pre-composition operation.

8.1.3. Conclusions. In many cases, the best results for time and memory are obtained in method NT (Compassion at the algorithmic level). Method NT is likely to be better for systems with big compassion sets. Method CJ requires an addition of a program variable for each compassion requirement. This decreases the performance of pre-composition operations, and is the reason that even though in some cases where the number of pre-composition operations is lower in method CJ, the performance is still worse since each pre-composition operations cost more.

By far, the worst results occured using method CA, where compassion is added as an antecedent of the property. The tester generated for such modified specifications has four additional variables, one for each principle temporal operator in the formula ($\Box \diamond p \rightarrow \Box \diamond q$) and also has additional justice requirements. These account for the decreased performance.

8.2. Reducing the number of justice conditions

When we define the FDS associated with a given program, we introduce a justice requirement for each of the program locations, to ensure that the process does not remain in that location indefinitely. However, this creates many justice conditions which may

```
  n
  ||  P[j] ::  [ ℓ0 : loop forever do       ]
 j=1           [     [ ℓ1 :  idle ]         ]
               [     [ ℓ2 :  skip ]         ]
               [     [  :         ]         ]
               [     [ ℓm :  skip ]         ]
```

Figure 11. Program CYCLE.

slow the algorithm down. In the following, we verify the effect of reducing the number of justice requirements, on the performance of algorithm FEASIBLE.

Figure 11 presents a toy example designed to allow us to create an FDS which corresponds to the program, such that the number of justice requirements is greatly reduced when compared to the FDS which would normally be generated. Each process of program CYCLE may remain at location ℓ_1 indefinitely. However, once the process executes ℓ_1 it must not remain stuck in one of the other program locations, rather, it should cycle through the rest of the program locations until it returns to ℓ_1.

The **idle** statement makes a nondeterministic choice between either remaining forever in the current program location, or advancing to the next statement. The **skip** statement does nothing except to advance to the next statement.

For each process i, program CYCLE has program locations $\ell_0, \ldots, \ell_m$. The standard FDS corresponding to program CYCLE contains, for each process i and for each $0 \leq j \leq m$, $j \neq 1$, a justice condition of the form $\neg at\_\ell_j$. Therefore, a program of p processes and m locations in each process has $p(m-1)$ justice conditions. In the modified FDS each process i has only a single justice condition:at_ℓ_1.

We verify the following accessibility property, for the first process:

$$\psi_3 : \Box(at\_\ell_2 \rightarrow \diamond at\_\ell_m)$$

Tables 6 and 7 compare executions of programs with 6 and 10 program locations. These tables show a slight improvement when less justice conditions are generated. However, program CYCLE was tailored to simplify the reduction of justice conditions. In other programs we have checked, such as DINE, the price of reducing the number of justice

Table 6. Six Program Locations.

N	5 Justice Cond.	1 Justice Cond.
10	0.57	0.26
14	4.74	4.44
18	12.57	11.62
22	26.90	26.24
26	49.90	49.58
30	89.49	87.60

Table 7. Ten program locations.

N	9 Justice Cond.	1 Justice Cond.
6	0.81	0.23
8	4.26	3.26
10	10.44	9.11
12	21.05	18.79
14	38.66	35.74
16	64.68	60.94

conditions was either adding a single, complex justice requirement, or adding variables to the verified system. In these cases, reducing the number of justice requirements increased execution time of the feasibility algorithm.

Note that for justice requirements of the form $\neg at\_\ell_i$, line 9 of algorithm FEASIBLE converges in only two steps. Although the standard, automatic way for generating an FDS from a program produces more justice requirements than what could optimally be produced by hand, the price for processing each of these justice requirement is small. This can explain our experimental results.

We thus conclude that reducing the number of justice conditions is usually not recommended.

References

1. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Inf. and Comp.*, Vol. 98, No. 2, pp. 142–170, 1992.
2. 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.*, Springer-Verlag, 1981, pp. 52–71.
3. E.M. Clarke, O. Grumberg, and K. Hamaguchi, "Another look at LTL model checking," *Formal Methods in System Design*, Vol. 10, No. 1, 1997.
4. E.M. Clarke, O. Grumberg, D.E. Long, and X. Zhao, "Efficient generation of counterexamples and witnesses in symbolic model checking," in *Proc. Design Automation Conference 95 (DAC95)*, 1995.
5. E.A. Emerson and C.L. Lei, "Efficient model-checking in fragments of the propositional modal μ-calculus," in *Proc. First IEEE Symp. Logic in Comp. Sci.*, pp. 267–278, 1986.
6. E.A. Emerson and C. Lei, "Modalities for model checking: Branching time logic strikes back," *Science of Computer Programming*, Vol. 8, pp. 275–306, 1987.
7. N. Francez, *Fairness*, Springer-Verlag, 1986.
8. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the temporal analysis of fairness," in *Proc. 7th ACM Symp. Princ. of Prog. Lang.*, pp. 163–173, 1980.
9. 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 and O. Grumberg (Eds.), *Proc. 9*th *Intl. Conference on Computer Aided Verification, (CAV'97), Volume 1254 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, 1997, pp. 268–278.
10. 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 Scandina vian Workshop on Algorithn Theory*, 1996, pp. 10–20.
11. R. Hojati, H. Touati, R.P. Kurshan, and R.K. Brayton, "Efficient ω-regular language containment," in G.V. Bochmann and D.K. Probst (Eds.), *Proc. 4*th *Intl. Conference on Computer Aided Verification (CAV'92),*

Volume 697 of Lect. Notes in Comp. Sci., Springer-Verlag, number 663 in Lect. Notes in Comp. Sci., SPringer-Verlag, 1992, pp. 396–409.

12. Y. Kesten and A. Pnueli, "Verification by augmented finitary abstraction, *Inf. and Comp.*, Vol. 163, pp. 203–243, 2000.
13. Y. Kesten, A. Pnueli, and L. Raviv, "Algorithmic verification of linear temporal logic specifications," in K.G. Larsen, S. Skyum, and G. Winskel (Eds.), *Proc, 25th Int. Col-loq. Aut. Lang. Prog.*, Volume 1443 of Lect. Notes in Comp. Sci., Springer-Verlag, 1998, pp. 1–16.
14. R.P. Kurshan, *Computer Aided Verification of Coordinating Processes*, Princeton University Press, Princeton, New Jersey, 1995.
15. O. Lichtenstein, "Decidability, completeness, and extensions of linear time temporal logic," PhD thesis, Weizmann Institute of Science, 1991.
16. O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, 1985, pp. 97–107.
17. 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., Springer-Verlag, 1981, pp. 264–277.
18. 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., Springer-Verlag, 1985, pp. 196–218.
19. Z. Manna and A. Pnueli, "Completing the temporal picture," *Theor. Comp. Sci.*, Vol. 83, No. 1, pp. 97–130, 1991.
20. Z. Manna and A. Pnueli, *Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, New York, 1991.
21. Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*, Springer-Verlag, New York, 1995.
22. A. Pnueli and E. Shahar, "A platform for combining deductive with algorithmic verification," in R. Alur and T. Henzinger, R. Alur and T. Henzinger (Eds.), *Proc. 8th Intl. Conference on Computer Aided Verification (CAV'96)*, Volume 1102 of Led. Notes in Comp. Sci., Springer- Verlag, 1996, pp. 184–195.
23. J.P. Queille and J. Sifakis, "Specification and verification of concurrent systems," in *cesar* in M. Dezani-Ciancaglini and M. Montanari (Eds.), *International Symposium on Programming*, Volume 137 of Lect. Notes in Comp. Sci., Springer-Verlag, 1982, pp. 337–351.
24. K. Ravi, R. Bloem, and F. Somenzi, "A comparative study of symbolic algorithms for the computation of fair cycles," in W.A. Hunt, Jr. and S.D. Johnson (Eds.), *Formal Methods in Computer Aided Design*, Volume 1954 of Lect. Notes in Comp. Sci., Springer-Verlag, 2000, pp. 143–160.
25. F.A. Stomp, W.-P. de Roever, and R.T. Gerth, "The μ-calculus as an assertion language for fairness arguments," *Inf. and Comp.*, Vol. 82, pp. 278–322, 1989.
26. M.Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proc. First IEEE Symp. Logic in Comp. Sci.*, 1986, pp. 332–344.
27. Z. Yang, "Performance analysis of symbolic reachability algorithms in model checking," Master's thesis, Rice University, 1999.