Model Checking with Strong Fairness *

Yonit Kesten† Amir Pnueli‡ Li-on Raviv‡ Elad Shahar‡

March 26, 2001

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 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 FDS's (equivalently, symbolically checking for the emptiness of Streett automata), based on the Emerson-Lei fixpoint characterization of both weak and strong fairness.

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.

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 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 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 Systems Engineering, Ben Gurion University, ykesten@bgumail.bgu.ac.il — Contact author

‡Weizmann Institute of Science, {amir,elad}@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 [LP84] 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 [LP84], 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 Streett automaton, we can view the algorithms presented in [EL85] and [LP84] as algorithms for checking the emptiness of Streett automata [VW86]. An improved algorithm solving the related problem of emptiness of Streett 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 [CGH97], which showed how to reduce model checking of a future PTL formula into CTL model checking. 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 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 paper we present an approach to the symbolic model checking of PTL 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 CTL model checking (as in [CGH97]) or to automata. 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 PTL 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 [CGH97] which, in turn, is an extension of the *statification* method used in [MP91a] and [MP95] 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 [LP84], identifies all computations satisfying a given set of fairness constraints. This involves the identification of all fair strongly connected components (SCC). However, while the enumerative algorithm identifies each SCC separately, the BDD-based symbolic algorithm is more efficient, identifying all states participating in some fair SCC simulataneously. Our symbolic algorithm can be viewed as a straightforward implementation of the nested fixed-point characterization of Emerson-Lei for fully fair computations [EL86], as opposed to the CTL model checkers which consider only the weak-fairness part of this characterization.

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 a given set of weak fairness constraints. The presented algorithm gives a heuristics which improves the performance of the Emerson-Lei algorithm. We use the same heuristics in our algorithm, while dealing with both types of fairness constraints.

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 Streett automata.

The symbolic feasibility algorithm presented here is not restricted to the treatment of PTL formulas. With minor modifications it can be applied to check the CTL formula $\mathbf{E}_{fair}\mathbf{G}p$, where the *fair* subscript refers now to full fairness. In [**?**], 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.

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 define the parallel composition of two FDSs into a single FDS. Both synchronous and asynchronous compositions are considered. In section 4 we present PTL, the propositional fragment of linear temporal logic, including the past operators. Next, in section 5 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 6, followed by an algorithm for extracting a witness (a counter example) in section 7. In section 8 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 9 with experimental results and final conclusions.

A (partial) conference version of this paper appeared in [KPR98].

2 Fair Discrete Structure

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 [MP91b] and [MP95]. 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.

- $V = \{u_1, ..., u_n\}$: 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 FDS. A state is defined to be *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. 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.

- $\mathcal{J} = \{J_1, \ldots, J_k\}$: A set of assertions expressing the *justice* requirements (also called *weak fairness requirements*). Intentionally, 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* requirements (also called *strong fairness requirements*). Intentionally, the compassion requirement for $\langle p, q\rangle \in \mathcal{C}$ 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 $\mathcal{D}$*-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 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 $\mathcal{D}$ be an FDS for which the above components have been identified. We define a *fair run* of $\mathcal{D}$ to be an infinite sequence of states $\sigma : s_0, s_1, s_2, ...,$ satisfying the following requirements:

- *Consecution:* For each $j = 0, 1, ...,$ the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.
- *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 must also contain infinitely many q-positions.

A fair run of is a *computation* of $\mathcal{D}$ if it satisfies the requirement

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.

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

3 Parallel Composition of FDS's

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

3.1 Asynchronous Parallel Composition

We define the *asynchronous parallel composition* of two FDS'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 $\mathcal{D}_1$ and $\mathcal{D}_2$ is a new system $\mathcal{D}$ whose basic actions are chosen from the basic actions of its components, i.e., $\mathcal{D}_1$ and $\mathcal{D}_2$. Thus, we can view the execution of $\mathcal{D}$ as the *interleaved execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$, and can use asynchronous composition in order to construct big concurrent systems from smaller components.

As seen from the definition, $\mathcal{D}_1$ and $\mathcal{D}_2$ may have different as well as common state variables, and the variables of $\mathcal{D}$ are the union of all of these variables. The initial condition of $\mathcal{D}$ is the conjunction of the initial conditions of $\mathcal{D}_1$ and $\mathcal{D}_2$. The transition relation of $\mathcal{D}$ states that at any step, we may choose to perform a step of $\mathcal{D}_1$ or a step of $\mathcal{D}_2$. 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 $\mathcal{D}_1$, we should preserve all variables in $V_2 - V_1$. The justice and compassion sets of $\mathcal{D}$ are formed as the respective unions of the justice and compassion sets of the component systems.

Asynchronous parallel composition is used to assemble an assynchronous system from its components.

3.2 Synchronous Parallel Composition

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

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

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 $\mathcal{D}_1$ 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 $\mathcal{D}_2$. Thus, we can view the execution of $\mathcal{D}$ as the *joint execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$.

In some cases, in particular when considering hardware designs which are naturally synchronous, we may 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* T_φ for an LTL property φ, as described in Section 5.

4 Linear Temporal Logic

As a requirement specification language for reactive systems we take the propositional fragment of *linear temporal logic* (PTL) [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 $\quad \ominus$ – Previous
$\mathcal{U}$ – Until $\quad \mathcal{S}$ – Since

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 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,
  $(\sigma, j) \models p \quad \Longleftrightarrow \quad s_j \models p$
  That is, we evaluate p locally, using the interpretation given by s_j.
- $(\sigma, j) \models \neg p \quad \Longleftrightarrow \quad (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \quad \Longleftrightarrow \quad (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \quad \Longleftrightarrow \quad (\sigma, j+1) \models p$
- $(\sigma, j) \models p \mathcal{U} q \quad \Longleftrightarrow \quad$ for some $k \geq j, (\sigma, k) \models q$,
  and for every i such that $j \leq i < k, (\sigma, i) \models p$
- $(\sigma, j) \models \ominus p \quad \Longleftrightarrow \quad j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p \mathcal{S} q \quad \Longleftrightarrow \quad$ 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 FDS $\mathcal{D}$, we can restrict our attention to the set of models which correspond to computations of $\mathcal{D}$, i.e., $\mathcal{C}omp(\mathcal{D})$. 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 P. 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:

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

5 Construction of Testers for PTL Formulas

In this section, we present the construction of a *tester* for a PTL 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

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

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

$$\begin{array}{rcl} \Theta_\varphi & = & \textit{past-init}(\varphi), \\ \text{where} & & \\ \textit{past-init}(\varphi) & = & \bigwedge_{\ominus p \in \varphi} \neg x_{\ominus p} \;\wedge\; \bigwedge_{p\mathcal{S}q \in \varphi} (x_{p\mathcal{S}q} \leftrightarrow \chi(q)) \end{array}$$

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 satisying q.

Note that, unlike the definition of testers presented in [KPR98, KP99], 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 6, for model checkin both LTL and CTL* properties over a finite FDS.

Transition Relation

The transition relation of T_φ is given by

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

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

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

Fairness Requirements

For each formula $p\mathcal{U}q \in \varphi$ which has a positive occurrence in φ (i.e., an occurrence under an even number of negations), we include in $\mathcal{J}$ the disjunction

$$\chi(q) \lor \neg x_{a\mathcal{U}q}$$

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 $\widetilde{\sigma}$ 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 $\sigma = s_0, \ldots$ satisfies the temporal formula φ iff σ is an X_φ-variant of a computation $\widetilde{\sigma} = \widetilde{s}_0, \ldots$ of T_φ, and $\widetilde{s}_0 \models \chi(\varphi)$.*

6 Checking for Feasibility

Let $\mathcal{D}$ be an FDS. We define a *run* of $\mathcal{D}$ to be a finite or infinite sequence of states which satisfy the requirements of initiality and consecution but not necessarily any of the justice or compassion requirements. We say that a state s is *$\mathcal{D}$-accessible* if it appears in some run of $\mathcal{D}$. When $\mathcal{D}$ is understood from the context, we simply say that state s is accessible. We say that $\mathcal{D}$ is *feasible* if $\mathcal{D}$ has at least one computation. We say that a state s is *$\mathcal{D}$-feasible*, if $\mathcal{D}$ has a computation $\sigma : s_0, s_1, s_2, \ldots$, such that $s = s_i$ for some $i \geq 0$.

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 [LP84] and [EL85] (for full explanations and proofs see [Lic91] and [MP95]). 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 $\mathcal{D}$-accessible 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 which is both just and compassionate.

For π, an infinite initialized path in $G_\mathcal{D}$, 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 $\mathcal{D}$ with fair subgraphs of $G_\mathcal{D}$.

Claim 2 *The infinite initialized path* π *is a computation of* $\mathcal{D}$ *iff* $\text{Inf}(\pi)$ *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:

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

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

The algorithm FEASIBLE presented in fig. 1, 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 $\rho_{\mathcal{D}}$, and compute in *new* the set of all accessible states.

The main loop contains three inner loops. The inner loop at lines 6–8 removes from *new* all states which are not R^*-predecessors of some J-state for all justice requirements $J \in \mathcal{J}$. Line 8 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 9–11, removes from *new* all p-states which are not R^*-predecessors of some q-state for some $(p, q) \in \mathcal{C}$. Line 11 restricts R again to pairs (s_1, s_2) where s_1 is currently in *new*.

Finally, the loop at lines 12–13, successively removes from *new* all states which do not have a succcessor in *new*.

Correctness of the Set-Based Algorithm

Let $\mathcal{D}$ be an FDS, s be a $\mathcal{D}$-accessible state and $U_{\mathcal{D}}$ be the set of states resulting from the application of algorithm FEASIBLE over $\mathcal{D}$. The following sequence of claims establish the correctness of the algorithm.

Algorithm FEASIBLE ($\mathcal{D}$) : **predicate** — Compute the set of $\mathcal{D}$-feasible states

new, old : **predicate**
R : **relation**

1. $old := \emptyset$
2. $R := \|\rho_{\mathcal{D}}\|$
3. $new := \|\Theta_{\mathcal{D}}\| \circ R^*$; $R := R \cap (new \times \Sigma)$
4. **while** $(new \neq old)$ **do**
   begin
5. $\quad old := new$
6. $\quad$ **for each** $J \in \mathcal{J}$ **do**
7. $\qquad new := R^* \circ (new \cap \|J\|)$
8. $\qquad R := R \cap (new \times \Sigma)$
9. $\quad$ **for each** $(p, q) \in \mathcal{C}$ **do**
   $\quad$ **begin**
10. $\qquad new := (new - \|p\|) \ \cup \ R^* \circ (new \cap \|q\|)$
11. $\qquad R := R \cap (new \times \Sigma)$
   $\quad$ **end**
12. $\quad$ **while** $(new \neq new \cap (R \circ new))$ **do**
13. $\qquad new := new \cap (R \circ new)$
14. $R := R \cap (new \times \Sigma)$
   end
15. **return**(new)

Figure 1: Algorithm FEASIBLE

Claim 3 (Termination) *The algorithm* FEASIBLE *terminates.*

Proof: 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 $R^* \circ new_0^4 = new_0^4$, it is not difficult 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 4 (Completeness) *If s is $\mathcal{D}$-feasible then $s \in U_{\mathcal{D}}$.*

Proof: Assume that s is $\mathcal{D}$-feasible. Then by definition, s is on an initialized infinite 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 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

$$\widetilde{S} : \quad S \cup \{s' \mid s' \text{ is on an initialized path to a state in } S\}$$

Obviously $s \in \widetilde{S}$. Following the operations performed by algorithm FEASIBLE, we can show that $\widetilde{S}$ 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, 10, and 13, cannot remove any state of

$\widetilde{S}$. Consequently, $\widetilde{S}$ must remain throughout the process and will be contained in $U_{\mathcal{D}}$, implying that $s \in U_{\mathcal{D}}$.

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

Proof: Assume that $s \in U_{\mathcal{D}}$. Let $S \subseteq U_{\mathcal{D}}$ be the set of states in $U_{\mathcal{D}}$, accessible from s. Since $s \in U_{\mathcal{D}}$ then s is $\mathcal{D}$-accessible. 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. 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 then it would consist of a single state that would have been removed on the last execution of the loop at lines 12-13. Let r be an arbitrary state within S_t. For every $J \in \mathcal{J}$, r can reach some J-state $\tilde{r} \in U_{\mathcal{D}}$ 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.

The Claims *Completeness* and *Soundness* lead to the following conclusion:

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

Corollary 7 *$\mathcal{D}$ is feasible iff the set $U_{\mathcal{D}} \cap \Theta_{\mathcal{D}} \neq \emptyset$.*

Proof: A direct result of Corollary 7 and the definition of feasibility.

The original enumerative algorithms of [EL85] and [LP84] 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, which is the approach taken in algorithm FEASIBLE, or symmetrically requiring closure under successors and looking for initial components. 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].

In addition to the choice between forward, backward and bi-directional closures, there are other possible variations of algorithm FEASIBLE. An experimental comparison of some of these variations is presented in section 9.

Model Checking of LTL Properties

Having presented an algorithm for verifying whether a finite-state FDS is feasible, we can now model check an LTL property φ over an FDS $\mathcal{D}$ as follows. Let $\Theta_{\mathcal{D},\varphi} = \Theta_{\mathcal{D}} \;\wedge\; \chi(\neg\varphi)$. Namely, $\Theta_{\mathcal{D},\varphi}$ is an assertion representing the set of states that satisfy the initial condition of $\mathcal{D}$ and the temporal formula $\neg\varphi$, in the combined FDS $\mathcal{D} \| T_{\neg\varphi}$. To model check the $\mathcal{D}$-validity $\mathcal{D} \models \varphi$,

- Construct the tester $T_{\neg\varphi}$.
- Construct the synchronous paralle composition $\mathcal{D} \| T_{\neg\varphi}$.
- Evaluate FEASIBLE$(\mathcal{D} \| T_{\neg\varphi}) \cap \Theta_{\mathcal{D},\varphi}$.

The verification is based on the following Claim.

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

The proof of equivalent claims can be found in [VW86], [LP84].

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 $D \models \varphi$ to checking the feasibility of $D \,|||\, T_{\neg\varphi}$, such a witness can be provided by a computation of the combined FDS $D \,|||\, T_{\neg\varphi}$.

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 $L_2 = (s_a, \ldots, s_b)$, 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. 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.

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 2: Function *path*.

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 D is feasible, it always has an *ultimately periodic* computation of the following form:

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

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

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

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

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

Figure 3: Algorithm WITNESS.

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

If D 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 $s \in \mathit{final}$, an arbitrarily chosen state within *final*. In the loop at lines 5 and 6 we search for a state s satisfying $R^* \circ \{s\} \subseteq \{s\} \circ R^*$. 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 moves 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 $\mathit{last}(\mathit{prefix})$ 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 the MSCS *final* contains some p-state. Finally, in line 17, we complete the path to form a closed cycle by looping back to $last(prefix)$.

8 Symbolic Model Checking of CTL* Properties

In the following, we show that algorithm FEASIBLE can be used for model checking 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 [EL87]. 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 $\mathcal{D}$, taking the full fairness constraints of $\mathcal{D}$ into consideration.

First we define the model of *fair computation structures,* which is the semantical model for CTL* formulas. Next, we present the syntax and semantics of CTL* with past operators (see [Eme90] for the future fragment of CTL*). Finally we present the symbolic model checking algorithm.

8.1 Fair Computation Structures

Let V be a finite set of variables. A *fair computation structure* over V is a tuple $\mathcal{K}_V : \langle S, S_0, R, J, C\rangle$, consisting of the following components.

- S : A (possibly infinite) set of all states over V.
- $S_0 \subseteq S$: A subset of *initial states*.
- $R \subseteq S \times S$: A transition relation, relating a state $s \in S$ to its $\mathcal{K}$-successor $s' \in S$.
- $J = (K_1, \ldots, K_k)$: A set of justice sets, where $K_i \subseteq S$ for every $i \in [1..k]$.
- $C = (\langle r_1, t_1\rangle, \ldots, \langle r_m, t_m\rangle)$: A set of compassion sets, where $r_i, t_i \subseteq S$ for every $i \in [1..m]$.

The set of justice and compassion sets is denoted the *fairness set*. Let $\pi : s_0, s_1, \ldots$ be an infinite sequence of states, and S be a set of states. We say that j is an S-position in π if $\pi_j \in S$, where π_j is the state at position j of π. Let $\mathcal{K}$ be a fair computation structure. The sequence $\pi = s_0, s_1, \ldots$ is said to be a *path in* $\mathcal{K}$[1] if it satisfies the following requirements:

- *Initiality:* $s_0 \in S_0$.
- *Consecution:* $(s_i, s_{i+1}) \in R$, for every $i \geq 0$.
- *Justice:* For every $K \in J$, π contains infinitely many K-positions,

[1] A path in a fair computation structure is defined as a fair path. A CTL* path formula is interpreted over a path in the structure, which is assumed to be a fair path. A similar approach is presented in [CGP99].

- *Compassion:* For every $\langle r, t\rangle \in C$, if π contains infinitely many r-positions, it must contain infinitely many t-positions.

8.2 The Logic CTL*

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.
  We refer to E_f and A_f as *path quantifiers*.
- 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 $p\mathcal{S}q$.

CTL* is the set of state formulas generated by the above rules.

The semantics of a CTL∗ formula p is defined with respect to a fair computation structure $\mathcal{K}$ over the vocabulary of p. The semantics is defined inductively as follows.
State formulas are interpreted over states in $\mathcal{K}$. We define the notion of a CTL* formula p holding at a state s in $\mathcal{K}$, denoted $(\mathcal{K}, s) \models p$, as follows:

- For an assertion p,
  $(\mathcal{K}, s) \models p \iff s \models p$
- $(\mathcal{K}, s) \models \neg p \iff (\mathcal{K}, s) \not\models p$
- $(\mathcal{K}, s) \models p \vee q \iff (\mathcal{K}, s) \models p$ or $(\mathcal{K}, s) \models q$
- $(\mathcal{K}, s) \models E_f p \iff (\mathcal{K}, \pi, j) \models p$ for some path $\pi \in \mathcal{K}$ and position j satisfying $\pi_j = s$.

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

- For an assertion p,
  $(\mathcal{K}, \pi, j) \models p \iff (\mathcal{K}, s) \models p$, for $s = \pi_j$
- $(\mathcal{K}, \pi, j) \models \neg p \iff (\mathcal{K}, \pi, j) \not\models p$
- $(\mathcal{K}, \pi, j) \models p \vee q \iff (\mathcal{K}, \pi, j) \models p$ or $(\mathcal{K}, \pi, j) \models q$
- $(\mathcal{K}, \pi, j) \models \bigcirc p \iff (\mathcal{K}, \pi, j+1) \models p$
- $(\mathcal{K}, \pi, j) \models p\mathcal{U}q \iff (\mathcal{K}, \pi, k) \models q$ for some $k \geq j$
  and $(\mathcal{K}, \pi, i) \models p$ for every i, $j \leq i < k$
- $(\mathcal{K}, \pi, j) \models \ominus p \iff (\mathcal{K}, \pi, j-1) \models p$
- $(\mathcal{K}, \pi, j) \models p\mathcal{S}q \iff (\mathcal{K}, \pi, k) \models q$ for some $k \leq j$
  and $(\mathcal{K}, \pi, i) \models p$ for every i, $j \geq i > k$

In the case that $(\mathcal{K}, s_0) \models p$, we say that the state formula p *holds* on $\mathcal{K}$, and denote it by $\mathcal{K} \models p$.

The notion of temporal validity (satisfiabilityty) of a CTL* formula is similar to that of an LTL formula. The notion of $\mathcal{D}$-satisfiability (validity) is also similar, as follows. Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be an FDS. Then $\mathcal{D}$ generates a fair computation structure $\mathcal{K}_V : \langle S, S_0, R, J, C\rangle$ as follows:

- The states of $\mathcal{K}$ are all the V-states of $\mathcal{D}$.
- $s \in S_0$ iff $s \models \Theta$.

- $(s_i, s_{i+1}) \in R$ iff $\rho(V_i, V_{i+1}) = \text{T}$.
- $s \in K_i$ iff $s \models J_i$.
- $s \in r_i$ (t_i) iff $s \models p_i$ $(s \models q_i)$.

A CTL* formula p is *$\mathcal{D}$-valid* if it holds over all initial states of $\mathcal{D}$.

8.3 Model Checking CTL*

Let $\mathcal{D}$ be a finite state FDS and $E_f\varphi$ be a CTL* formula with no embedded path quantifiers. Figure 4 presents algorithm SAT-E_f which uses algorithm FEASIBLE to evaluate the set of $\mathcal{D}$-accessible states satisfying the formula. Note that, since φ is a CTL* formula with no embedded path quantifiers, it is also an LTL formula, for which a tester can be constructed.

Algorithm SAT-E_f $(\mathcal{D}, \varphi)$: **predicate** — Set of $\mathcal{D}$-accessible states satisfying $E_f\varphi$

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

Figure 4: Algorithm SAT-E_f

The following claim assert that the set $S_{\mathcal{D},E\varphi}$ evaluated by algorithm SAT-E_f is exactly the set of $\mathcal{D}$-accessible states satisfying the CTL*formula $E_f\varphi$.

Claim 9 $(\mathcal{D}, s) \models E_f\varphi \iff s \in S_{\mathcal{D},E\varphi}$

Similarly, let $\mathcal{D}$ be an FDS and $A_f\varphi$ be a CTL* formula with no embedded path quantifiers. To evaluate the set of $\mathcal{D}$-accessible states satisfying the formula, we use the CTL* congruence

$$A_f\varphi \approx \neg E_f\neg\varphi.$$

Using ealgorithm SAT-E_f we first evaluate $E_f\neg\varphi$, then negate the resulting assertion.
Finally, to model check an arbitrary CTL* formula p, namely, a CTL* formula with embedded path quantifiers, consider the formula from right to left, evaluating a single path quantifier at a time. The result of each step is an assertion $(S_{\mathcal{D},E\varphi})$ used to evaluate the next (embedding) CTL* formula. Note that at each step, we evaluate a single CTL* formula with no embedded path quantifiers.

9 Experimental Results

The algorithms described in this paper have been implemented within the TLV system [PS96]. In the following section, we summarize our experimental results, all related to the main algorithm, namely algorithm FEASIBLE.

The experiments were carried on a Sun Ultra with 1 Gigabyte of memory. The programs used for experimentation are *parametric* programs. In the following tables summarizing our results, timing results are given in seconds and the columns titled N indicate the number of processes for which the programs have been tested.

9.1 Compassion at the Algorithmic Level

To test the advantage of dealing with compassion at the algorithmic level, we consider the program DINE presented in Fig. 5. This program is a simple solution to the dining philosophers problem, using semaphores for coordination between processes.

Program DINE satisfies the safety requirement of mutual exclusion (no neighboring philosophers can dine at the same time), however, this naive algorithm fails to satisfy the liveness requirement of *accessibility* for the first process:

$$\psi_1 : \square(at\_\ell_2 \rightarrow \diamondsuit\, at\_\ell_4),$$

which states that when the first philosopher wishes to dine it will eventually do so. To ensure accessibility, we add two compassion requirements for each of the processes, associated with locations ℓ_2 and ℓ_3 .

$$\begin{array}{l}
\begin{array}{lll} \textbf{in} & n & : \textbf{integer where } n \geq 2 \\ \textbf{local} & c & : \textbf{array } [1..n] \textbf{ where } c = 1 \end{array} \\[1em]
\overset{n}{\underset{j=1}{\Big|\Big|}} \; P[j] :: \left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_1 : & \textbf{noncritical} \\ \ell_2 : & \textbf{request } c[j] \\ \ell_3 : & \textbf{request } c[j \oplus_n 1] \\ \ell_4 : & \textbf{critical} \\ \ell_5 : & \textbf{release } c[j] \\ \ell_6 : & \textbf{release } c[j \oplus_n 1] \end{array} \right] \end{array} \right]
\end{array}$$

Figure 5: Program DINE: The Dining Philosophers.

We checked the property of accessibility over program DINE, using algorithm FEASIBLE in which compassion is dealt with at the algorithmic level. The verification attempt failed because program DINE does not satisfy the property of accessibility. We then compared the results of these runs with two other methods used with verification algorithms that do not permit compassion at the algorithmic level. The first, transforms the compassion requirements into an LTL formula which is added as antecedent to the verified property. Let φ be the property we wish to verify. Let $\mathcal{C}$ be the set of compassion requirements. The new property to be verified is:

$$\left(\bigwedge_{\langle p,q \rangle \in \mathcal{C}} (\square \diamondsuit p \rightarrow \square \diamondsuit q) \right) \rightarrow \varphi$$

This case is considered in subsection 9.1.1.

The second method transforms compassion requirements into justice, and is discussed in subsection 9.1.2.

9.1.1 Compassion as Antecedent to the Verified Property

Each process of program DINE contributes two compassion requirements, due to program locations ℓ_2 and ℓ_3. Table 1 summarizes the results of verifying accessibility for program DINE with 3 and 4 processes. The column headed DINE represents the case in which all compassion requirements are dealt with at the algorithmic level. Column DINE-1 (DINE-2) represents the cases that one (two) of the compassion requirements for each of the proceeses is transformed into an LTL formula, added as an antecedent to the verified property.

The advantage of dealing with compassion at the algorithmic level is clear.

N	DINE	DINE-1	DINE-2
3	0.18	0.45	1.73
4	0.66	4.02	51.08

Table 1: Compassion as Antecedent to the Verified Property

9.1.2 Transforming Compassion into Justice

Every compassion requirement can be 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:

$$\begin{array}{lcl}
\widetilde{V} & : & V \cup \{r_i\} \\
\widetilde{\Theta} & : & \Theta \wedge (r_i = \mathrm{F}) \\
\widetilde{\rho} & : & \rho \wedge (r \rightarrow (\neg p \wedge r')) \\
\widetilde{\mathcal{J}} & : & \mathcal{J} \cup \{(r \vee q)\} \\
\widetilde{\mathcal{C}} & : & \mathcal{C} - \{(p_i, q_i)\}
\end{array}$$

Table 2 compares the result of verifying the accessibility property for program DINE with various number of processes. Column DINE represents the results of algorithm FEASIBLE, with all compassion requirements dealt with at the algorithmic level. Column DINE-1 (DINE-2) presents the result with one (two) of the compassion requirements of each process transformed into justice requirement. Again, handling the compassion requirements explicitly is preferable according to this test.

N	DINE	DINE-1	DINE-2
3	0.18	0.24	0.34
4	0.65	1.18	1.82
5	3.24	6.02	12.02
6	17.14	39.69	120.95

Table 2: Compassion Transformed into Justice

9.2 Forward, Backward and Bi-directional

Algorithm FEASIBLE finds closures under predecessors. We call this direction of searching *backward*. It is possible to modify algorithm FEASIBLE to find closures under successors (forward) or both successors and predecessors (bi-directional). Note that under the forward analysis, the set $U_{\mathcal{D}}$ is a

$$
\begin{array}{lll}
\textbf{in} & n & : \textbf{integer where } n \geq 2 \\
\textbf{local} & ch & : \textbf{array}[1..n] \textbf{ of channel}[1] \textbf{ of } [0..2] \\
\textbf{local} & A & : \textbf{array}[1..n] \textbf{ of boolean where } A = 1
\end{array}
$$

$$
\overset{n}{\underset{j=1}{\Big\|}} P[j] :: \left[
\begin{array}{l}
\mathit{changed} : \ \textbf{boolean where } \mathit{changed} = 0 \\[1ex]
M :: \left[
\begin{array}{l}
\ell_0 : \ \textbf{loop forever do} \\
\quad \left[
\begin{array}{ll}
\ell_1 : & \textbf{Active} \\
\ell_2 : & A[j] := 0 \\
\ell_3 : & \textbf{when } \bigvee_{i=1}^{n} A[i] \textbf{ do } (A[j], \mathit{changed}) := (1,1)
\end{array}
\right]
\end{array}
\right] \\[2ex]
\qquad \| \\[2ex]
C :: \left[
\begin{array}{l}
\begin{array}{ll}
\mathit{first} : & \textbf{boolean where } \mathit{first} = (j = 1) \\
\mathit{token} : & [0..2] \textbf{ where } \mathit{token} = 0
\end{array} \\[1ex]
m_0 : \ \textbf{loop forever do} \\
\quad \left[
\begin{array}{ll}
m_1 : & \textbf{if } \mathit{first} \\
 & \textbf{then} \quad m_2 : \ \mathit{first} := 0 \\
 & \textbf{else} \quad m_3 : \ ch[j] \Rightarrow \mathit{token} \\
m_4 : & \textbf{if } j = 1 \wedge \mathit{token} = 2 \\
 & \textbf{then} \quad m_5 : \ \textbf{Terminate} \\
m_6 : & \textbf{if } j = 1 \\
 & \textbf{then} \quad m_7 : \ \mathit{token} := \mathit{token} + 1 \\
m_8 : & \textbf{await } \neg A[j] \\
m_9 : & \left\langle
\begin{array}{rcl}
ch[j \oplus_n 1] & \Leftarrow & \max(\mathit{token} - \mathit{changed}, 1) \\
\mathit{changed} & := & 0
\end{array}
\right\rangle
\end{array}
\right]
\end{array}
\right]
\end{array}
\right]
$$

Figure 6: Program TERM-DET: Termination Detection in a Distributed Dystem.

subset of the set of $\mathcal{D}$-feasible states, however, algorithm FEASIBLE still satisfies Corollary 7. The following is a comparison of these three variants of algorithm FEASIBLE.

In figure 6 we present program TERM-DET, which detects termination in a distributed system [DFvG83]. Each process in program TERM-DET consists of two submodules. Module M is the *main* part of the process. Statement **Active** is a schematic statement representing the normal activity of the process. At any point, module M may decide to become inactive by moving to location ℓ_2. On becoming inactive process $P[j]$ sets variable $A[j]$ to 0, signaling that it enters an inactive phase. An inactive process may be reactivated. Statement ℓ_3 allows the process to become non-deterministically active, provided there is still some active process in the system.

Module C is responsible for the termination detection mechanism. The n processes are arranged in a ring spanned by the array of channels $ch[1], \ldots, ch[n]$. Process $P[1]$ initiates the detection process by sending its right neighbor a token 1 on channel $ch[2]$ as soon as $P[1]$ becomes inactive. Each process receiving the token sends it to its right neighbor when the process becomes inactive.

The token of 1 eventually should come back to process $P[1]$. When this token arrives, process $P[1]$ sends a token of 2 (computed at m_6) to $P[2]$ and this new token starts another circle around the ring. Each process maintains a local boolean variable *changed* which is initially 0. This variable is set to 1 whenever a process changes its status from inactive to active, and is cleared to 0 when a process sends its token to a right neighbor. Thus at any point, *changed* $= 1$ implies that the process was reactivated since the last time it sent the token. If a process receives a token of value 2, yet it was reactivated since the last time it sent the token, it sends a token of 1 instead. When $P[1]$ receives back a token of 2, it decides that all the processes are inactive and the system has terminated. This is signified by process $P[1]$ moving to location m_5.

We verify the liveness property stating that when all processes are inactive (i.e. at location ℓ_3) the first process moving to m_5, detecting termination:

$$\psi_2 : \Box((\bigwedge_{j=1}^{n} \neg A[j]) \rightarrow \Diamond(at\_m_5[1]))$$

Table 3 presents the results of comparing forward, backward and bi-directional closure in algorithm FEASIBLE, verifying the property ψ_2 over program TERM-DET. In Table 4 we present the results of a similar comparison for the accessibility property ψ_1 over program DINE.

In these tests forward and bi-directional directions have the best performance.

N	Forward	Backward	Bi-directional
2	0.58	0.36	0.35
3	2.22	2.08	1.42
4	9.90	11.02	6.06
5	49.63	51.76	31.78

Table 3: Termination Detection for Program TERM-DET

N	Forward	Backward	Bi-directional
4	0.66	2.70	1.00
5	3.19	14.89	5.16
6	17.27	95.56	28.13

Table 4: Accessibility for Program DINE

9.3 Removing States with no Successors

In lines 12–13 of algorithm FEASIBLE, all states with no successors in the set of states *new*, are removed at each external iteration. However, this may be too exhaustive. If the external loop is executed many times and the strands of such states are short, these states will be removed even if we just remove the first state of such strands at each iteration.

To test this we checked a version of algorithm FEASIBLE, where lines 12–13 are replaced by a single execution of the body of the loop (line 13).

Tables 5 and 6 compare this new version of algorithm FEASIBLE to the original one, for both programs DINE and TERM-DET. In both examples, the original algorithm has a better performance.

N	Exhaustive	One step
3	2.32	2.14
4	10.20	11.26
5	51.01	53.67

Table 5: Program TERM-DET

N	Exhaustive	One step
4	0.65	2.64
5	3.20	14.79
6	17.15	95.32

Table 6: Program DINE

9.4 Restriction

In lines 8, 11 and 14 of algorithm FEASIBLE the transition relation R is intersected with the set of states *new*, restricting R to transitions that enter a state $s \in new$. However, it seems that the this restriction of R is mainly needed at line 8, and could be avoided at lines 11 and 14.

Tables 7 and 8 compare the restriction strategies, for programs TERM-DET and DINE accordingly. For program TERM-DET the original algorithm, with more restriction, performed significantly better. For program DINE the unrestricted version performed slightly better. This can be explained by the cost of the restriction operation when the restriction is not beneficial.

N	Less Restriction	Original
2	0.46	0.57
3	2.99	2.22
4	30.04	9.79
5	868.26	50.67

Table 7: Termination detection for Program TERM-DET

N	Less Restriction	Original
4	0.62	0.65
5	3.12	3.19
6	16.07	17.08
7	111.55	122.20

Table 8: Accessibility for Program DINE

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

We use the following simple CYCLE program. Each process cycles through m program locations. All of the program locations except the first, have an associated justice condition.

For each process i, program CYCLE has program location $\ell_1, \ldots, \ell_m$. The original system contains, for each process i and for each $2 \leq j \leq m$, a justice condition of the form $\neg at\_\ell_j$. Therefore, a program of m program locations and p processes has $p*(m-1)$ justice conditions. In the modified system each process i has a single justice condition: at_ℓ_1.

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

$$\psi_3 : \square(at\_\ell_2 \rightarrow \diamondsuit\, at\_\ell_m)$$

Tables 9 and 10 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 conditions was either adding a single, complex justice requirement, or adding variables to the verified system. In these cases, the result of reducing the number of justice requirements were slowing down the feasibility algorithm.

Table 9: 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 10: 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

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

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*, pages 52–71. Lec. Notes in Comp. Sci. 131, Springer-Verlag, 1981.

[CGH97] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1), 1997.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[DFvG83] E.W. Dijkstra, W.H. Feijen, and A.J.M van Gasteren. Derivation of a terminatio detection algorithm for disrtibued computations. *Info. Proc. Lett.*, 16:217–219, 1983.

[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.

[EL86] 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.*, pages 267–278, 1986.

[EL87] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[Eme90] E.A. Emerson. Temporal and modal logics. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, volume B, pages 995–1072. Elsevier, 1990.

[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.

[KP99] Yonit Kesten and Amir Pnueli. Verification by augmented finitary abstraction. *Inf. and Comp.*, 1999. A special issue. To appear.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 1–16. Springer-Verlag, 1998.

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

[LP84] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 11th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1984.

[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.*, pages 264–277. Lec. Notes in Comp. Sci. 115, 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.