Formal Methods in System Design, 1: 297–322 (1992)

A Stubborn Attack on State Explosion

ANTTI VALMARI
Technical Research Centre of Finland, Computer Technology Laboratory, PO Box 201, SF-90571 Oulu, Finland

Abstract. This article presents the *LTL-preserving stubborn set method* for reducing the amount of work needed in the automatic verification of concurrent systems with respect to linear-time temporal logic specifications. The method facilitates the generation of *reduced state spaces* such that the truth values of linear temporal logic formulas are the same in the ordinary and reduced state spaces. The only restrictions posed by the method are 1) the formulas must be known before the reduced state-space generation is commenced; 2) the use of the temporal operator "next state" is prohibited; and 3) the (reduced) state space of the system must be finite. The method cuts down the number of states by utilizing the fact that in concurrent systems the net result of the occurrence of two events is often independent of the order of occurrence.

Keywords: temporal logic, verification, state-space reduction

1. Introduction

The automatic verification of temporal properties of finite-state systems has been a topic of intensive research during the recent decade. A typical approach is to generate the state space of the system and then apply a model-checking algorithm on it to decide whether the system satisfies given temporal logic formulas [1, 2]. A well-known problem of this approach is that the state spaces of systems tend to be very large, rendering the verification of medium-size and large nontrivial systems impossible with a realistic computer. This problem is known as the *state explosion* problem.

Concurrency is a major contributor to state explosion. It introduces a large number of execution sequences that lead from a common start state to a common end state by the same transitions, but the transitions occur in different order, causing the sequences to go through different states. This phenomenon was recognized long ago, and the choice of a coarser level of atomicity has been suggested as a partial solution (see [3]). Unfortunately, the power of coarsening the level of atomicity is limited. Consider a system consisting of n processes that execute k steps without interacting with each other and then stop. The system has $(k+1)^n$ states. Each of the processes of the system can be coarsened to a single atomic action. Coarsening reduces the number of states to 2^n, which is still exponential in the number of the processes [4]. However, it seems intuitively that to check various properties of the system it would be sufficient to simulate the processes in one arbitrarily chosen order, thus generating only $nk+1$ states.

To our knowledge, the first person to suggest a concurrency-based state-space reduction method potentially capable of changing a state space from exponential to polynomial in the number of processes was W. Overman [5]. Overman's work is little known, perhaps because he considered a very restricted case (the terminal states of systems consisting of processes that do not branch or loop), and the algorithm he gave as part of his method for finding certain sets was not efficient enough from the practical point of view. He suggested also a modified method with a faster algorithm, but the modification destroyed the ability of changing exponential state spaces to polynomial.

The problems in Overman's approach were essentially solved by Valmari when he presented the so-called *stubborn set* method [6, 7]. The method is presented in the framework of ordinary Petri nets in [6]. In [7], some reduction power is sacrificed to present the theory in a more general *variable/transition* framework that is directly applicable to concurrent programs, etc. Overman's method can be thought of as a special case of the more powerful and more widely applicable stubborn set method as presented in [7]. The computational problem that caused Overman to modify his method now reappeared as the problem of finding "good" *stubborn sets*. In [6], a stubborn set search algorithm is provided that (if required) finds the sets in Overman's method and, assuming that the system is loosely coupled, is linear in the size of the system under analysis. However, the sets found by the algorithm (including Overman's sets) are not necessarily the "best" stubborn sets in the sense of leading to the best state-space reduction results. In [7], a quadratic (under reasonable assumptions) stubborn set search algorithm is developed that is capable of finding, in a certain sense, "best" stubborn sets. These results are summarized, compared to Overman's work, and refined in some details in [4].

The above-mentioned articles treat properly only the detection of deadlocks and terminal states using the reduced state spaces, although they anticipate the analysis of more properties. Their key theorem is "every reachable state without enabled transitions is present in the reduced state space." It is shown in [8] that the basic stubborn set method also preserves the existence of infinite execution sequences. It is thus suitable for detecting nontermination. Furthermore, the article essentially gains back the part of the reduction power lost in the shift from [6] to [7].

The difference between the theories in [6, 8] and [7] is clarified by the distinction of the *weak* and *strong* theories of stubborn sets in [9]. The weak theory is more complicated and more difficult to implement, but it leads to better reduction results. The above-mentioned linear stubborn set search algorithm cannot exploit the extra freedom offered by the weak theory, but the quadratic algorithm can. In [9], the way was also opened to the use of stubborn sets in the verification of other than termination-oriented properties by solving the so-called *ignoring problem*. As a concrete result, the article shows how the reachability of a state satisfying a given state predicate can be decided using stubborn sets. This renders possible the verification of system invariants.

The stubborn set method was applied to colored Petri nets in [10], and an initial attempt to apply it to process algebras was made in [11].

The present article extends the stubborn set method to almost full linear temporal logic—almost, because the operator "next state" is forbidden. The resulting method is called the *LTL-preserving stubborn set method.* For simplicity, we have chosen to use the strong stubborn set framework, although with minor refinements the results are valid in the weak theory as well. An earlier version of this article was published as [12].

P. Godefroid and P. Wolper have developed a theory of *trace automata*, which resembles the stubborn set theory but is based on somewhat different thinking. Trace automata resemble reduced state spaces obtained with the stubborn set method. In [13], Godefroid presents an algorithm for constructing trace automata, and uses trace automata to check whether the language generated by one Petri net is a subset of the language generated by the other. The method gives very good reduction but relies on quite strict assumptions about one of the nets. The trace automaton construction algorithm in [13] is incorrect, but the errors have been fixed in the final version of [14].

Trace automata are used in [15] to check the validity of linear temporal logic formulas; therefore, the goal of [15] is the same as the goal of this article. However, the method is different. In [15] the formula is represented by a Büchi automaton that is connected in parallel with the system under analysis. Therefore, the formula guides the construction of the trace automaton. In this way, the generation of some parts of the trace automaton that do not affect the validity of the formula can be avoided. This is sometimes a great advantage, and it is often referred to as "on-the-fly" verification. The condition that [15] gives for deciding from the trace automaton whether the formula is valid is incorrect. The authors of the article have recently pointed out that it can, however, be used as a partial test, which may give the answer "yes," "no," or "cannot say." The method in [15] is more restricted than the method in the present article in that it assumes that certain processes of the system cannot ever be blocked. We call this the assumption of "nonblocking processes." If this assumption cannot be guaranteed, then the method cannot be used. On the other hand, the assumption allows the representation of fairness assumptions, which is an advantage over the method in this article. Fairness assumptions can, however, be assigned only to nonblocking processes.

Trace automata are used for the checking of safety properties in [14]. The new contribution of the article is a technique for checking whether a state satisfying a given state predicate is reachable. This technique is based on transforming the problem to the deadlock detection problem by adding extra transitions and states to the system. This is different from [9], where the same problem was solved by monitoring the reduced state space. To detect deadlocks, the article uses a simplified and corrected version of the trace automaton algorithm in [13]. (The corrections are included only in the final version of [14]; the algorithm in the Participants' Version of the Proceedings is still incorrect.) Excluding so-called

"sleep sets," the corrected deadlock detection method is a special case of the method in [6]. The sleep sets are an important contribution by the authors, and they were mentioned already in [13].

Other works having some ideas in common with the stubborn set method are [16–19]. In [16], very strong restrictions are posed on the underlying computational model, due to which the order in which processes perform their next transitions becomes insignificant. Therefore, it becomes possible to limit the analysis to global transitions consisting of the simultaneous execution of one transition from each process. In [17], the behavior of a concurrent system is represented by an *optimal simulation*. An optimal simulation is a minimal—in a certain sense representative—collection of executions in which the ordering of independent successive events is left unspecified by (in essence) executing them simultaneously. The article develops a graph representation for optimal simulations. The method presented in [18] represents a concurrent fragment of the execution of a system in a symbolic form by employing a new process algebraic operator. A logic that facilitates proving the properties of all interleavings of concurrent events by investigating only some interleavings is presented in [19].

In this article, the theory underlying the LTL-preserving stubborn set method is developed in section 2. Section 3 discusses how the theory may be implemented, and section 4 contains an example.

2. Definitions and basic theorems

2.1. *Variable/transition systems*

To develop the stubborn set method, we look at a concurrent system as a system consisting of a finite set $\mathbb{V}$ of *variables* and a finite set $\mathbb{T}$ of *transitions*. Each variable v has an associated set called *type* and denoted by *type*(v), and at every instant of time v has a unique *value* belonging to its type. Assuming an ordering of $\mathbb{V}$, the Cartesian product of the types of the variables is the set of *syntactic states* and is denoted by $\mathbb{S}$. The value of variable v at syntactic state s is denoted by $s(v)$. There is a partial *next state* function *next* from $\mathbb{S} \times \mathbb{T}$ to $\mathbb{S}$ that defines when a transition is *enabled* and what is the result of the *occurrence* of an enabled transition. Transition t is enabled in state s, denoted by *en*(s,t), iff *next*(s,t) is defined. If *next*(s,t) = s' we say that t may *occur* at s producing s' and often write $s \xrightarrow{t} s'$. We often merge the states between successive transition occurrences and write $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} s_n$ instead of $s_0 \xrightarrow{t_1} s_1 \wedge s_1 \xrightarrow{t_2} s_2 \wedge \dots \wedge s_{n-1} \xrightarrow{t_n} s_n$. A sequence like this is called a *finite execution sequence* and its *length* is n. The *concatenation* of the execution sequences $\sigma = s_0 \xrightarrow{t_1} \dots \xrightarrow{t_n} s_n$ and $\rho = r_0 \xrightarrow{d_1} \dots \xrightarrow{d_m} r_m$ where

$s_n = r_0$ is defined by $\sigma^\circ\rho = s_0 \ -t_1\rightarrow \ \ldots \ -t_n\rightarrow \ s_n t d_1 \ldots \ -d_m\rightarrow \ r_m$.

The relation "$\rightarrow$" is defined by $s \rightarrow s' \Leftrightarrow \exists t \in \mathbb{T} : s \ -t\rightarrow \ s'$, and "$\rightarrow^*$" is the reflexive transitive closure of "$\rightarrow$". There is a distinguished state $\mathbb{s}_0$ called the *initial state* of the system. A *variable/transition system* or *v/t system* is the 5-tuple $(\mathbb{V}, \mathbb{T}, \mathit{type}, \mathit{next}, \mathbb{s}_0)$, where the components of the 5-tuple are as just explained.

Section 4.1 contains a nontrivial example of a v/t system.

The stubborn set method relies on the analysis of certain relationships between transitions. Let us define and explain some necessary concepts.

Definition 1. Let $t, t' \in \mathbb{T}, V \subseteq \mathbb{V}$ and $T \subseteq \mathbb{T}$.

- t is *enabled with respect to* V at $s \in \mathbb{S}$, denoted by $\mathit{en}(s, t, V)$, iff $\exists\, s' \in \mathbb{S} : \mathit{en}(s', t) \wedge \forall v \in V : s'(v) = s(v)$.
- T is a *write-up* set of t with respect to V, iff for every $t' \in \mathbb{T}$ and every $s, s' \in \mathbb{S}$ $\neg \mathit{en}(s, t, V) \wedge s \ -t'\rightarrow \ s' \wedge \mathit{en}(s', t, V) \Rightarrow t' \in T$.
- t *accords with* t', denoted by $t \leftrightarrow t'$, iff for every $s \in \mathbb{S}$ $\mathit{en}(s, t) \wedge \mathit{en}(s, t') \Rightarrow \exists\, s', s_1, s_1' \in \mathbb{S} : s \ -t\rightarrow \ s_1 \ -t'\rightarrow \ s_1' \wedge s \ -t'\rightarrow \ s' \ -t\rightarrow \ s_1'$.

The intuition behind the definition of $\mathit{en}(s, t, V)$ is perhaps best understood by noticing that if t is *not* enabled with respect to V, then it is necessary to modify the value of at least one variable in V to enable t. The definition has the following rather obvious consequence: $\mathit{en}(s, t) \Leftrightarrow \mathit{en}(s, t, \mathbb{V}) \Leftrightarrow \forall V \subseteq \mathbb{V} : \mathit{en}(s, t, V)$.

A write-up set of the transition t with respect to the set V of variables is any set of transitions containing at least the transitions that have the potential of modifying the status of t from disabled with respect to V to enabled with respect to V. We do not require the write-up set to be the smallest such set, because minimality is not needed in the theory of stubborn sets, and the smallest set may be difficult to find in practice. For instance, if the specification of transition t' contains a writing reference to a variable in V but has a complicated enabling condition evaluating to `false`, t' does not belong to the smallest write-up set of t with respect to V because it is never enabled. However, it may be difficult to see that t' can be ruled out. Our definition allows the use of a write-up set that can be easily computed and is an upwards approximation of the minimal set. Every transition t and subset of variables V has at least one write-up set, namely, the set of all transitions $\mathbb{T}$. From now on we assume that a unique write-up set is defined for every (t, V)-pair. We denote it by $\mathit{wrup}(t, V)$.

The definition of "according with" can be illustrated graphically (see figure 1). "According with" is a static, symmetric commutativity property of transition pairs. Concurrent systems typically contain several pairs of transitions according with each other. Let us denote the set of variables tested in the enabling condition of a transition t or read or written during the execution of t by $\mathit{ref}(t)$. The set $\mathit{ref}(t)$ is called the *reference set* of t. Two transitions accord with each other if their

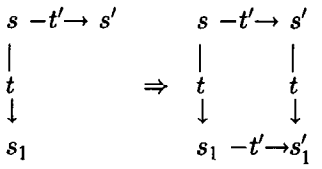

Figure 1. Definition of "according with."

reference sets are disjoint. The same is true even if there are common variables in the reference sets, as long as the transitions never write to them. Transitions writing to a fifo queue accord with transitions reading from it, unless there are other variables in common. Transitions corresponding to different locations in the code of a sequential process accord with each other independently of the variables to which they refer, because they are never simultaneously enabled.

The notion of reference set is extended to variables and sets of variables as follows.

Definition 2. Let $v \in \mathbb{V}$ and $V \subseteq \mathbb{V}$.

- $ref(v) = \{t \in \mathbb{T} \mid v \in ref(t)\}$
- $ref(V) = \cup_{v \in V} ref(v)$

2.2. *Stubborn sets*

We are ready to define *semistubborn* sets of transitions:

Definition 3. Let $T \subseteq \mathbb{T}$ and $s \in \mathbb{S}$. T is *semistubborn* at s, iff $\forall t \in T$:

1. $\neg en(s,t) \Rightarrow \exists V \subseteq \mathbb{V} : \neg en(s,t,V) \wedge wrup(t,V) \subseteq T$
2. $en(s,t) \Rightarrow \forall t' \notin T : t \leftrightarrow t'$

Part 1 of the definition guarantees that a disabled transition belonging to a semistubborn set can be enabled only by the transitions in the set. Part 2 says that enabled transitions in a semistubborn set accord with outside transitions. It is not difficult to prove that at least the empty set and the set of all transitions $\mathbb{T}$ are semistubborn at every state, and thus semistubborn sets always exist. Furthermore, if a transition outside a semistubborn set occurs, the set remains semistubborn. This is a justification for the peculiar term semi*stubborn*. The significance of semistubborn sets is in the following theorem:

$$\begin{array}{ccccccccc} s_0 & -t_1\rightarrow & s_1 & -t_2\rightarrow & \ldots & -t_{n-2}\rightarrow & s_{n-2} & -t_{n-1}\rightarrow & s_{n-1} \\ \downarrow t_n & & & & & & & & \downarrow t_n \\ s'_0 & -t_1\rightarrow & s'_1 & -t_2\rightarrow & \ldots & -t_{n-2}\rightarrow & s'_{n-2} & -t_{n-1}\rightarrow & s'_{n-1} \\ & & & & & & & & = s_n \end{array}$$

Figure 2. Illustration of theorem 1.

Theorem 1. Let $T \subseteq \mathbb{T}$ and $s_0 \in \mathbb{S}$ such that T is semistubborn at s_0. Let $n \geq 1$, and let

$$\sigma = s_0 \ -t_1\rightarrow \ s_1 \ -t_2\rightarrow \ \ldots \ -t_{n-2}\rightarrow \ s_{n-2} \ -t_{n-1}\rightarrow \ s_{n-1} \ -t_n\rightarrow \ s_n$$

be a finite execution sequence such that $t_1, \ldots, t_{n-1} \notin T$ and $t_n \in T$. There is the finite execution sequence

$$\sigma' = s_0 \ -t_n\rightarrow \ s'_0 \ -t_1\rightarrow \ s'_1 \ -t_2\rightarrow \ \ldots \ -t_{n-2}\rightarrow \ s'_{n-2} \ -t_{n-1}\rightarrow \ s'_{n-1},$$

where $s'_{n-1} = s_n$.

The theorem can be illustrated graphically. The sequence σ corresponds to the top and right edges of figure 2, while σ' corresponds to the left and bottom sides.

The proof is a straightforward application of the definitions, and can be found in [7], [8], and [9].

Theorem 1 allows us to move the occurrence of a transition belonging to a semistubborn set from the future to the current state. However, this is of course of no use if no transition in the semistubborn set is going to occur in the future. This is certainly the case if the set is empty. Therefore, we augment the definition by the requirement that there must be an enabled transition in the set.

Definition 4. Let $T \subseteq \mathbb{T}$ and $s \in \mathbb{S}$. T is *stubborn* at s, iff T is semistubborn at s, and $\exists\, t \in T$: $en(s,t)$.

A stubborn set exists exactly when there is an enabled transition, because then the set of all transitions $\mathbb{T}$ is stubborn. However, as will soon become obvious, it is advantageous (but not mandatory) to use stubborn sets containing as few enabled transitions as possible. It is possible to check from definitions 3 and 4 that the union of stubborn sets is stubborn.

2.3. Reduced state spaces

A variable/transition system defines a labeled directed graph called its *state space* in a natural way:

Definition 5. The *state space* of the *v/t system* ($\mathbb{V}$, $\mathbb{T}$, *type*, *next*, s_0) is the triple ($\mathbb{W}$, $\mathbb{E}$, $\mathbb{T}$), where

- $\mathbb{W} = \{s \in \mathbb{S} \mid s_0 \rightarrow^* s\}$
- $\mathbb{E} = \{(s, t, s') \in \mathbb{W} \times \mathbb{T} \times \mathbb{W} \mid s \text{ -}t\rightarrow s'\}$

Let TS be a function from $\mathbb{S}$ to the set of the subsets of $\mathbb{T}$ such that $TS(s)$ is stubborn if $\exists t \in \mathbb{T} : en(s, t)$, and $TS(s) = \emptyset$ otherwise. We call the function TS a *stubborn set generator*. The stubborn set method uses a stubborn set generator to generate a *reduced state space* as follows. The generation starts at s_0. Assume that s has been generated. In ordinary state-space generation, every transition enabled at s is used to generate the immediate successor states of s. In the stubborn set method, only the enabled transitions in $TS(s)$ are used. If $TS(s)$ contains less enabled transitions than $\mathbb{T}$, the number of immediate successors is reduced. This often leads to a reduction in the total number of states. To distinguish between reduced and ordinary state-space concepts, we use underlining notation as follows:

- $\underline{en}(s, t) \Leftrightarrow en(s, t) \wedge t \in TS(s)$
- $s \text{ -}t\underline{\rightarrow}\ s' \Leftrightarrow s \text{ -}t\rightarrow s' \wedge t \in TS(s)$
- $s\underline{\rightarrow}s' \Leftrightarrow \exists t \in TS(s) : s \text{ -}t\rightarrow s'$
- "$\underline{\rightarrow}^*$" is the reflexive and transitive closure of "$\underline{\rightarrow}$"

Definition 6. Assume that a stubborn set generator TS is given. The *reduced state space* of the v/t system ($\mathbb{V}, \mathbb{T}, type, next, s_0$) is the triple ($\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T}$), where

- $\underline{\mathbb{W}} = \{s \in \mathbb{S} \mid s_0\underline{\rightarrow}^* s\}$
- $\underline{\mathbb{E}} = \{(s, t, s') \in \underline{\mathbb{W}} \times \mathbb{T} \times \underline{\mathbb{W}} \mid s \text{ -}t\underline{\rightarrow}\ s'\}$

The definition implies that $\underline{\mathbb{W}} \subseteq \mathbb{W}$ and $\underline{\mathbb{E}} \subseteq \mathbb{E}$. The ordinary state space is a special case of a reduced state space, because we may choose $TS(s) = \mathbb{T}$ for states with enabled transitions. However, our goal is to keep the reduced state space small, to save effort in its generation and in the model checking afterwards. Perhaps surprisingly, always choosing the stubborn set with the smallest number of enabled transitions does not necessarily lead to the smallest reduced state space [4]. However, it is obvious that if T_1 and T_2 are stubborn and the set of enabled transitions in T_1 is a proper subset of the corresponding set of T_2, then T_1 is preferable. We say that a stubborn set is *optimal* if it is the best possible in this respect. In [6] and [7], a linear and a quadratic algorithm are given for

finding almost optimal and optimal stubborn sets, respectively. (The complexity claims assume that the systems are loosely coupled.) The linear algorithm is particularly attractive because its best-case complexity is better than linear; if it finds a stubborn set close to its starting point, it optimizes it as much as it can and stops without investigating the rest of the v/t system. The linear algorithm is described in more detail in section 3.

2.4. Execution sequences in reduced state spaces

This section is devoted to a construction that, given a finite execution sequence of the system under analysis, finds an execution sequence that is present in the reduced state space and is, roughly speaking, a permutation of an extension of the former. The construction is in the heart of most proofs in the stubborn set theory. It proceeds in steps. Each step appends a transition occurrence to the end of the constructed sequence. The transition occurrence is either picked and removed from the original sequence, in which case we say that an original transition occurrence is *consumed*, or a fresh new transition occurrence is found for the purpose. The construction may be continued until the original sequence is exhausted. It may happen that only fresh transitions are used from some step onwards, in which case the construction may be continued endlessly.

The construction is presented formally below. The original finite execution sequence is denoted by σ. The constructed sequence after step i is denoted by $\underline{\sigma}_i$ and its last state by $\underline{s}_i$. The execution sequences ρ_i and σ_i correspond to the unconsumed part of the original sequence and the original sequence appended by the occurrences of the fresh transitions used by the construction. The symbol $k(i)$ denotes the length of ρ_i, that is, the number of still unconsumed transition occurrences. It may be helpful to notice that $\underline{\sigma}_i \circ \rho_i$ exists and that its first and last state are the same as the first and last state of σ_i. Furthermore, the transition occurrences of $\underline{\sigma}_i \circ \rho_i$ are the same as the transition occurrences of σ_i, but not necessarily in the same order.

Construction 1. Let TS be a stubborn set generator and $\sigma = s_0 -t_1\rightarrow \ldots -t_n\rightarrow s_n$ be a finite execution sequence. The states $\underline{s}_i$ and execution sequences $\underline{\sigma}_i$, σ_i, and $\rho_i = r_{0,i} -d_{1,i}\rightarrow \ldots -d_{k(i),i}\rightarrow r_{k(i),i}$ are defined recursively as follows. The symbol $k(i)$ is defined as the length of ρ_i.

- $\sigma_0 = \rho_0 = \sigma$ and $\underline{\sigma}_0 = s_0 = \underline{s}_0$.
- If $k(i) = 0$, that is, if ρ_i consists of one state and no transition occurrences, the construction cannot be continued.
- If $k(i) > 0$, then $d_{1,i}$ is enabled at $r_{0,i}$, and thus $TS(r_{0,i})$ is stubborn. There are two cases.
  1. ρ_i contains at least one occurrence of a transition belonging to $TS(r_{0,i})$. We define $k'(i) = k(i), \sigma'_i = \sigma_i$ and $\rho'_i = \rho_i$.

2. ρ_i contains no occurrences of transitions belonging to $TS(r_{0,i})$. By definition 4, $TS(r_{0,i})$ contains an enabled transition t'_i. By part 2 of definition 3, t'_i is enabled at $r_{1,i}, \ldots, r_{k(i),i}$. We define $k'(i) = k(i) + 1, d_{k(i)+1,i} = t'_i, \sigma'_i = \sigma_i -t'_i\rightarrow r_{k'(i),i}$, and $\rho'_i = \rho_i -t'_i\rightarrow r_{k'(i),i}$.

   By construction, ρ'_i contains at least one occurrence of a transition belonging to $TS(r_{0,i})$. Let $1 \leq j(i) \leq k'(i)$ be chosen such that $d_{1,i}, \ldots, d_{j(i)-1,i} \notin TS(r_{0,i})$ and $d_{j(i),i} \in TS(r_{0,i})$. By theorem 1, there is the sequence $\rho''_i = r_{0,i} -d_{j(i),i}\rightarrow r'_{0,i} -d_{1,i}\rightarrow \ldots -d_{j(i)-1,i}\rightarrow r_{j(i),i} -d_{j(i)+1,i}\rightarrow \ldots -d_{k'(i),i}\rightarrow r_{k'(i),i}$. By the $(i-1)$th construction step, $r_{0,i} = \underline{s}_i$. Hence, $d_{j(i),i} \in TS(\underline{s}_i)$, and we may define $\underline{s}_{i+1} = r'_{0,i}, \underline{\sigma}_{i+1} = \underline{\sigma}_i -d_{j(i),i}\rightarrow \underline{s}_{i+1}$, and $\sigma_{i+1} = \sigma'_i$. The sequence ρ_{i+1} is defined by $\rho''_i = r_{0,i} -d_{j(i),i}\rightarrow \rho_{i+1}$. After this ith construction step, $r_{0,i+1} = r'_{0,i} = \underline{s}_{i+1}$.

In the future we will need the fact that in the above construction $d_{x,i} = d_{x,i+1}$ for $x = 1, \ldots, j(i) - 1$. Also the order of transition occurrences in $\underline{\sigma}_{i+1}{}^{\circ}\rho_{i+1}$ is otherwise the same as the order of transition occurrences in $\underline{\sigma}_i{}^{\circ}\rho_i$, but either the occurrence of $d_{j(i),i}$ $(1 \leq j(i) \leq k(i))$ has moved to a different location, or a new transition occurrence $d_{k(i)+1,i}$ has been introduced.

2.5. *Linear temporal logic preservation theorem*

In this section we state and prove the theorem underlying the linear temporal logic- (LTL) preserving stubborn set method. LTL formulas state properties of infinite sequences of states. Infinite execution sequences of a v/t system give naturally rise to infinite sequences of states. We also consider *stopping* execution sequences that are finite and that end at a state without enabled transitions. As usual, we extract infinite sequences of states from stopping execution sequences by letting the last state repeat forever. We say that the infinite or stopping execution sequence σ *satisfies* the LTL formula φ iff φ is a true statement about the infinite sequence of states extracted from σ. We say that φ is *valid* in state s in a given state space, iff there is no infinite or stopping execution sequence σ in the state space starting at s such that σ satisfies $\neg\varphi$.

Let $\Phi = \{\varphi_1, \ldots, \varphi_f\}$ be a collection of LTL formulas. We associate with Φ the set *vis*(Φ) of *visible transitions*. Transition t is *properly visible*, iff there are syntactic states s and s' such that $s -t\rightarrow s'$ and the truth value of at least one state predicate appearing in at least one formula in Φ is different at s and s'. The set *vis*(Φ) is some fixed set containing *at least* the properly visible transitions. The reason why we allow in *vis*(Φ) the presence of transitions that are not properly visible is the same reason we allow the write-up set to be an upwards approximation of the smallest set with the required property. Transition t is *visible* if $t \in$ *vis*(Φ); otherwise, t is *invisible*. The property we will use in the future is that when an invisible transition occurs, the truth values of the state predicates in the formulas in Φ do not change.

The LTL-preserving stubborn set method works only with *stuttering-invariant* formulas. Informally, stuttering-invariance means that the truth value of a formula on an infinite sequence of states does not change if one or more or even all of the states of the sequence are multiplicated, where multiplication means the replacement of the state by a positive finite number of copies of the state. Among the common LTL operators, "next state" ($\bigcirc$) and "previous state" may introduce formulas that are not stuttering-invariant. Formulas containing no other temporal operators than "henceforth" ($\Box$), "eventually" ($\Diamond$), "until" ($\mathcal{U}$), and their corresponding past operators and the operators derived from them are stuttering-invariant (see [20]).

We call the sequence of states $s_0 s_1 \ldots s_n$ an *elementary cycle*, if $n > 0, s_0 = s_n, s_i \neq s_j$ when $0 \leq i < j < n$ and $s_i \rightarrow s_{i+1}$ when $0 \leq i < n$. We can now formulate the theorem underlying the LTL-preserving stubborn set method:

Theorem 2. Let $(\mathbb{V}, \mathbb{T}, \mathit{type}, \mathit{next}, \mathbb{s}_0)$ be a v/t system and $\mathbb{S}$ and $(\mathbb{W}, \mathbb{E}, \mathbb{T})$ its set of syntactic states and its state space, respectively. Let Φ be a collection of stuttering-invariant LTL formulas such that the domain of the state predicates in the formulas is $\mathbb{S}$. Let TS be a stubborn set generator and $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$ the corresponding reduced state space such that the following hold:

1. $\underline{\mathbb{W}}$ is finite.
2. For every $\underline{s} \in \underline{\mathbb{W}}$, either
   a. $TS(\underline{s})$ contains no enabled visible transitions, or
   b. $\mathit{vis}(\Phi) \subseteq TS(\underline{s})$.
3. For every $\underline{s} \in \underline{\mathbb{W}}$, if there is an enabled invisible transition, then $TS(\underline{s})$ contains an enabled invisible transition.
4. Every elementary cycle of $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$ contains at least one state $\underline{s}$ such that $\mathit{vis}(\Phi) \subseteq TS(\underline{s})$.

Claim: $\varphi \in \Phi$ is valid at $\mathbb{s}_0$ in $(\mathbb{W}, \mathbb{E}, \mathbb{T})$ if and only if φ is valid at $\mathbb{s}_0$ in $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$.

Proof. It is sufficient to show that $(\mathbb{W}, \mathbb{E}, \mathbb{T})$ contains an execution sequence σ that starts at $\mathbb{s}_0$ and satisfies $\neg\varphi$ if and only if $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$ contains an execution sequence $\underline{\sigma}$ with the same properties. The "if" part is obvious, since $\underline{\mathbb{W}} \subseteq \mathbb{W}$ and $\underline{\mathbb{E}} \subseteq \mathbb{E}$. To prove the "only if" part, we construct from σ an execution sequence $\underline{\sigma}$ such that $\underline{\sigma}$ starts at $\mathbb{s}_0$ and the sequences of the occurrences of visible transitions in σ and $\underline{\sigma}$ are the same. Then the claim follows from the facts that σ satisfies $\neg\varphi$ and φ is stuttering-invariant. We consider three cases. Let $\underline{s}_i$, σ_i, ρ_i, $\underline{\sigma}_i$, $j(i)$, $d_{1,i}, \ldots, d_{j(i),i}$, and t'_i be defined as in construction 1.

1. σ is stopping. Let n be its length. We apply construction 1 n steps on σ to get $\underline{\sigma} = \underline{\sigma}_n$. Assume inductively that $0 \leq i < n, \sigma_i = \sigma$, and the sequence of the occurrences of visible transitions in $\underline{\sigma}_i{}^\circ \rho_i$ is the same as in σ. Because no transition is enabled at s_n (the last state of σ and σ_i), branch 2 of construction 1

cannot be taken. Because $i < n$, some branch of construction 1 can be taken. So branch 1 is taken, and $\sigma_{i+1} = \sigma$. By induction $\sigma_n = \sigma$ and $\rho_n = s_n$. Because of assumption 2 of theorem 2, if $d_{j(i),i}$ is visible, then $vis(\Phi) \subseteq TS(\underline{s}_i)$; thus $d_{1,i}, \ldots, d_{j(i)-1,i}$ are invisible. As a result, the sequence of the occurrences of visible transitions in $\underline{\sigma}_{i+1} \circ \rho_{i+1}$ is the same as in $\underline{\sigma}_i \circ \rho_i$. The same is obviously true if $d_{j(i),i}$ is invisible. We conclude that the sequence of the occurrences of visible transitions in $\underline{\sigma}_n = \underline{\sigma}_n \circ \rho_n$ is the same as in σ. Furthermore, $\underline{\sigma}_n$ is stopping.

2. σ is infinite and contains a finite number of visible transition occurrences. Let $|\underline{\mathbb{W}}|$ be the number of states in the reduced state space (finite by assumption 1 of theorem 2), and let m be the length of the prefix of σ up to and including the last occurrence of a visible transition. Let $\sigma' = s_0 -t_1\rightarrow \ldots -t_n\rightarrow s_n$ be a prefix of σ of length $n = (m+1)|\underline{\mathbb{W}}|$. We apply construction 1 n times on σ' to generate $\underline{\sigma}_1, \ldots, \underline{\sigma}_n$.

   Consider first $0 \leq i < n$ such that ρ_i contains the occurrence of a visible transition. As above, assumption 2 of theorem 2 guarantees that if $d_{j(i),i}$ is visible, then $d_{1,i}, \ldots, d_{j(i)-1,i}$ are invisible. This implies that branch 1 of the construction must be taken. As a result, the sequence of the occurrences of visible transitions in $\underline{\sigma}_{i+1} \circ \rho_{i+1}$ is the same as in $\underline{\sigma}_i \circ \rho_i$, and thus the same as in σ. The same is obviously true if $d_{j(i),i}$ is invisible.

   Suppose still that ρ_i contains the occurrence of a visible transition $d_{v(i),i}$. Remember that by construction 1, $d_{x,i} = d_{x,i+1}$ for $x = 1, \ldots, j(i) - 1$. Furthermore, the construction can be continued until all transition occurrences in ρ_i are consumed. Assume for a while that none of $d_{1,i}, \ldots, d_{v(i),i}$ is consumed during the next $|\underline{\mathbb{W}}|$ construction steps, that is, $j(i), j(i+1), \ldots, j(w-1) > v(i)$ and $d_{1,i} = \ldots = d_{1,w}$ and $\ldots$ and $d_{v(i),i} = \ldots = d_{v(i),w}$, where $w = i + |\underline{\mathbb{W}}|$. Consider the $|\underline{\mathbb{W}}| + 1$ states $\underline{s}_i, \ldots, \underline{s}_w$. They must contain an elementary cycle of $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$. By assumption 4 of theorem 2, there is $h \in \{i, \ldots, w-1\}$ such that $vis(\Phi) \subseteq TS(\underline{s}_h)$. Because $d_{v(i),h} = d_{v(i),i} \in vis(\Phi)$, by construction 1 $j(h) \leq v(i)$, a contradiction. We conclude that there is $0 \leq M \leq m|\underline{\mathbb{W}}|$ such that the occurrences of visible transitions in σ' are consumed by exactly M construction steps. That is, $\rho_0, \ldots, \rho_{M-1}$ contain an occurrence of a visible transition, but ρ_M does not. Thus the sequence of the occurrences of visible transitions in $\underline{\sigma}_M$ is the same as in σ. Note that so far we have not used the assumption about the length of σ'.

   Consider (at last!) $M \leq i < n$ such that ρ_i does not contain an occurrence of a visible transition. Because $i < n$, ρ_i contains a transition occurrence. There is thus an enabled invisible transition at $\underline{s}_i$. By assumption 3, construction 1 can be continued so that $d_{j(i),i}$ is invisible. If we do so, the sequence of the occurrences of visible transitions in $\underline{\sigma}_i$ is the same as in σ when $M \leq i \leq n$. Because $M \leq n - |\underline{\mathbb{W}}|$, at least the last $|\underline{\mathbb{W}}|$ transition occurrences in $\underline{\sigma}_n$ correspond to invisible transitions. Therefore, $\underline{\sigma}_n$ has a prefix $\underline{\sigma}_e = \underline{\sigma}_v \circ \underline{\mu}$ such that $M \leq v < e \leq n$, the sequence of the occurrences of visible transitions in $\underline{\sigma}_v$ is the same as in σ, $\underline{\mu}$ does not contain occurrences of visible transitions,

and $\underline{s}_e = \underline{s}_v$. We define $\underline{\sigma} = \underline{\sigma}_v{}^\circ\underline{\mu}^\circ\underline{\mu}^\circ\underline{\mu}^\circ\ldots$.

3. σ contains an infinite number of visible transition occurrences. Let σ_i be a prefix of length i of σ. We saw in case 2 of this proof that there is $0 \leq M(i) \leq i|\underline{\mathbb{W}}|$ such that all occurrences of visible transitions in σ_i are consumed by exactly $M(i)$ steps of construction 1. Let us denote the resulting sequence by $\underline{\sigma}_i$. By case 2, the sequence of the occurrences of visible transitions in $\underline{\sigma}_i$ is the same as in σ_i. We apply the reasoning of the proof of König's lemma to show that there is an infinite execution sequence $\underline{\sigma}$ such that the sequence of the occurrences of visible transitions in it is the same as in σ. Let $\Omega_0 = \{0, 1, \ldots\}$ and $\underline{s}_0 = \mathbb{s}_0$. Because $\mathbb{T}$ is finite, there is a transition t_1 and a state $\underline{s}_1$ such that $\underline{\sigma}_i$ is of the form $\underline{s}_0 -t_1\rightarrow \underline{s}_1 \ldots$ for an infinite set $\Omega_1 \subseteq \Omega_0$ of different values of i. Similarly, there are t_2 and $\underline{s}_2$ such that $\underline{\sigma}_i$ is of the form $\underline{s}_0 -t_1\rightarrow \underline{s}_1 -t_2\rightarrow \underline{s}_2 \ldots$ for an infinite set $\Omega_2 \subseteq \Omega_1$ of different values of i. The same reasoning may be continued to produce t_3, $\underline{s}_3$, and Ω_3, and so on forever. We define $\underline{\sigma}$ as the resulting infinite execution sequence $\underline{s}_0 -t_1\rightarrow \underline{s}_1 -t_2\rightarrow \ldots$. Let t_v be a visible transition occurring at position v in σ. By case 2, t_v is consumed by at most $v|\underline{\mathbb{W}}|$ construction steps. Thus the position of t_v in $\underline{\sigma}_v, \underline{\sigma}_{v+1}, \ldots$ and, consequently, in $\underline{\sigma}$ is at most $v|\underline{\mathbb{W}}|$. Thus $\underline{\sigma}$ contains the same visible transition occurrences and in the same order as σ.

3. Implementation of the method

To implement the LTL-preserving stubborn set method, it is necessary to find a practical way of generating reduced state spaces such that assumptions 1 to 4 of theorem 2 hold. Then the validity of the formulas in Φ can be checked using the reduced state space instead of the ordinary one. The use of the stubborn set method is thus transparent to the model-checking phase.

Assumption 1 is certainly satisfied if the system under analysis has a finite number of states because $\underline{\mathbb{W}} \subseteq \mathbb{W}$. If $\mathbb{W}$ may be infinite, then the LTL formula-validity decision problem is generally undecidable, and it is not surprising if an analysis method does not work. In some cases, though, $\underline{\mathbb{W}}$ may be finite even if $\mathbb{W}$ is not. An example of this is the simple system consisting of a producer and a consumer connected by an unbounded fifo buffer. The stubborn set method may alternate the buffer read and write operations (they accord with each other) in such a way that items do not accumulate in the buffer, if the state predicates of Φ do not test the content of the buffer.

In section 3.1 we describe an algorithm that can be used to implement a stubborn set generator TS such that $TS(s)$ satisfies assumptions 2 and 3. Section 3.2 shows how the reduced state-space generation may be organized so that assumption 4 is also satisfied.

3.1. *Constructing stubborn sets*

Assumptions 2a and 3 of theorem 2 are trivially satisfied if no transition is enabled in the current state s. Therefore, throughout the remainder of this subsection we assume that s has an enabled transition. We start by discussing the problem of finding "good" stubborn sets without worrying about assumptions 2 and 3 for the moment.

From a v/t system and its state s we derive a finite directed graph Γ_s using the definition of semistubborn sets (definition 3) as follows. The vertices of Γ_s correspond one-to-one to the transitions of the system. For simplicity, the vertex corresponding to $t \in \mathbb{T}$ is denoted by t. We say that a vertex is *enabled* if the corresponding transition is. Consider an enabled transition t. If t belongs to a semistubborn set T, then part 2 of definition 3 implies that every transition t' such that $\neg(t \leftrightarrow t')$ belongs to T. This is represented in Γ_s by introducing from t to an edge for every t' with the above property. Now, let t be disabled. Assume that $V \subseteq \mathbb{V}$ is chosen such that $\neg en(s, t, V)$. We require that $\forall t' \in wrup(t, V) : t' \in T$ and represent it again by introducing an edge from t to every t'. The graph Γ_s is now ready.

Let *reachable*(t) be the set of transitions reachable from t in Γ_s. Due to the construction of Γ_s, a set of transitions T is semistubborn if (but not necessarily only if) $\forall t \in T : reachable(t) \subseteq T$. If t_e is enabled, then *reachable*(t_e) is stubborn.

On the basis of the above, a stubborn set can be constructed by finding an enabled transition t_e and computing *reachable*(t_e). However, such stubborn sets may be much larger than necessary. If t_1 and t_2 are enabled, $t_2 \in reachable(t_1)$ but $t_1 \notin reachable(t_2)$, then *reachable*$(t_2)$ is stubborn and a proper subset of *reachable*(t_1). Obviously, *reachable*(t_2) is better than *reachable*(t_1). What we want to find is a stubborn set such that all its enabled transitions are reachable from each other. Such a stubborn set contains a maximal strongly connected component (strong component, for short) C of Γ_s such that C contains all the enabled transitions of the set and no other strong component containing enabled transitions is reachable from C. The set C^* defined as *reachable*(t) for any $t \in C$ is the stubborn set we are after. However, in practice it is sufficient to return C instead of C^*, since we do not need the disabled transitions of a stubborn set.

Strong components can be found in time linear in the size of Γ_s using Tarjan's algorithm [21]. Tarjan's algorithm finds strong components in depth-first order. Therefore, when Tarjan's algorithm for the first time announces a strong component C containing an enabled transition, the strong components reachable from C have already been investigated; they contain no enabled transitions. This shows the correctness of the following algorithm.

Algorithm 1. Let Γ_s be the graph defined above. Run Tarjan's algorithm on Γ_s until it finds a strong component C containing an enabled transition. Return C.

Algorithm 1 is guaranteed to find a stubborn set if there is an enabled transition.

It was first presented in [6]. To estimate its time complexity, we assume that the system is *loosely coupled* in the following sense: there is a small constant c such that $|ref(t)| \leq c$ for every $t \in \mathbb{T}$, and $|ref(v)| \leq c$ for every $v \in \mathbb{V}$. This assumption is reasonable when the stubborn set method is used, because the stubborn set method usually does not give good reduction results and is thus not worth using when there is no concurrency, i.e. when the system is tightly coupled. Furthermore, we assume that when t is disabled, a "reasonably small" set $V \subseteq ref(t)$ such that $\neg en(s, t, V)$ can be found in $O(|ref(t)|)$ time. In practice, V is found based on the syntactic structure of the system, as will be demonstrated by the example in section 4.1. It is actually legal to choose $V = ref(t)$ if a smaller V cannot be found fast enough. However, to obtain best reduction results it is advantageous to choose a small V. With these assumptions, if t is enabled, the number of edges of the form (t, t') in Γ_s is at most c^2. Because $V \subseteq ref(t)$, we have $|V| \leq c$ and $|wrup(t, V)| \leq c|V| \leq c^2$. So, when t is disabled, there are again at most c^2 edges of the form (t, t'). As a consequence, Γ_s has $|\mathbb{T}|$ vertices and at most $c^2|\mathbb{T}|$ edges. So the time consumed by algorithm 1 is at most of the order of $c^2|\mathbb{T}|$.

It is not uncommon in practice that algorithm 1 investigates only a small portion of Γ_s. Thus it is not reasonable to construct Γ_s in its entirety at each state. Instead, it should be constructed as needed by algorithm 1. In this way, the best-case complexity of algorithm 1 becomes sublinear.

In the derivation of algorithm 1, we did not specify how the $V \subseteq \mathbb{V}$ such that $\neg en(s, t, V)$ should be chosen when t is disabled. Algorithm 1 produces stubborn sets independent of how the choices are made, but the choices may affect the sizes of the stubborn sets. In this respect the stubborn sets produced by the algorithm are not necessarily optimal. In [7], a different, typically $O(c^2|\mathbb{T}|^2)$ stubborn set search algorithm is discussed that is capable of optimizing the choices. Algorithm 1 produces stubborn sets that are minimal with respect to set inclusion for the choices of V used in it.

Algorithm 1 can be used as a building block of an algorithm finding stubborn sets satisfying assumptions 2 and 3 of theorem 2. If algorithm 2 (below) terminates in step 1, then it returns the essential parts C of a stubborn set C^* such that C^* does not contain enabled visible transitions. Therefore, because C^* contains an enabled transition, it contains an enabled invisible transition. Thus C^* satisfies assumptions 2a and 3 of theorem 2. By construction, the sets T_u and $T_u \cup C^*$ produced by steps 2 and 3 contain an enabled transition and have the property $\forall t \in T : reachable(t) \subseteq T$, so they are stubborn. If algorithm 2 terminates in step 2 or 3, assumption 2b is satisfied due to the fact that the returned set has T_u as a subset. The C constructed in step 3 contains an enabled invisible transition, because during its construction algorithm 1 operates on a graph from which T_u and thus all visible transitions have been, in essence, removed. Thus assumption 3 is ensured by step 3 if the assumption is not valid already after step 2.

Algorithm 2

1. Apply algorithm 1 starting at every so far uninvestigated enabled invisible transition until either
   a. the strong component C returned by algorithm 1 contains no enabled visible transitions; or
   b. there are no more uninvestigated enabled invisible transitions.

   If "a" holds, then return C and terminate.
2. Compute the union of *reachable*(t) of every visible t and denote it by T_u. If T_u contains an enabled invisible transition, or if there are no enabled invisible transitions, then return T_u and terminate.
3. Apply algorithm 1 starting at an enabled invisible transition such that the search does not enter any of the transitions in T_u. Return $C \cup T_u$, where C is the result of the application of algorithm 1. □

Step 2 can be implemented by any common graph traversal algorithm, so steps 2 and 3 of algorithm 2 can be performed in $O(c^2|\mathbb{T}|)$ time. Step 1 consists of repeated applications of algorithm 1. It thus appears to be of the complexity $O(c^2|\mathbb{T}|^2)$. However, applications of algorithm 1 need not enter the parts of Γ_s investigated by earlier applications, if the information whether an enabled visible transition is reachable from the current transition is backward propagated and stored with the transitions. In this way, step 1 also, and consequently algorithm 2 as a whole, can be implemented in $O(c^2|\mathbb{T}|)$ time.

3.2. Ensuring assumption 4

Assumptions 2 and 3 of theorem 2 are local in the sense that they can be ensured at each state separately. This is different from assumption 4, which talks about the properties of certain kind of sets of states. We ensure it by generating the reduced state space in a certain order, by continuously monitoring the validity of assumption 4, and by taking recovery action when assumption 4 is violated. How this is done is shown by algorithm 3.

Algorithm 3

Main Program:
/* Calls *traverse* until all states have been investigated, i.e., expanded. This is needed because *traverse* is not guaranteed to expand all states it encounters. */

$(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T}) := (\{s_0\}, \emptyset, \mathbb{T})$; *search-stack* := $\emptyset$;
while $\underline{\mathbb{W}}$ contains an unexpanded state **do**
    choose an unexpanded state s from $\underline{\mathbb{W}}$ and perform *traverse*(s)
endwhile

Procedure *traverse*(s):
/* Generates and investigates a part of the reduced state space. This procedure is based on a depth-first search on the graph which is obtained by removing the output edges of so-called "green" states from $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$. State s is green if $vis(\Phi) \subseteq TS(s)$, s has no output edges, or $TS(s)$ contains all enabled transitions. This search discipline enables *traverse* to detect violations of assumption 4 and to recover from them by calling *re-expand* when necessary. */

expand(s);
if s is red **then**
    search-stack := *search-stack* $\cup \{s\}$;
    if there are $t \in \mathbb{T}$ and $s' \in$ *search-stack* such that $(s,t,s') \in \underline{\mathbb{E}}$ **then**
    /* red cycle test */
        re-expand(s) /* recovery action */
    else
        for each $t \in \mathbb{T}$ and $s' \in \underline{\mathbb{W}}$ such that $(s,t,s') \in \underline{\mathbb{E}}$ and s' is unexpanded
            do *traverse*(s')
        endfor
    endif
    search-stack := *search-stack* $- \{s\}$
endif

Procedure *expand*(s):
/* Constructs the output edges of s and determines the initial color of s. */

mark s expanded; mark s green;
if s has an enabled transition **then**
    construct a set T using algorithm 2;
    $\underline{\mathbb{W}} := \underline{\mathbb{W}} \cup \{s' \mid \exists t \in T : s \xrightarrow{t} s'\}$;
    $\underline{\mathbb{E}} := \underline{\mathbb{E}} \cup \{(s,t,s') \mid t \in T \wedge s \xrightarrow{t} s'\}$;
    if $vis(\Phi) \not\subseteq T \wedge T$ does not contain all enabled transitions **then**
        mark s red
    endif
endif

Procedure *re-expand*(s):
/* Paints a red state s green and modifies the output edges of s accordingly. */

apply step 2 of algorithm 2 to produce T_u;
$\underline{\mathbb{W}} := \underline{\mathbb{W}} \cup \{s' \mid \exists t \in T_u : s \xrightarrow{t} s'\}$;
$\underline{\mathbb{E}} := \underline{\mathbb{E}} \cup \{(s,t,s') \mid t \in T_u \wedge s \xrightarrow{t} s'\}$;
mark s green

The main loop of the algorithm and the definitions of *expand* and *re-expand* suffice to guarantee that if the algorithm terminates, it produces a reduced state space $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$ such that assumptions 2 and 3 of theorem 2 are satisfied. This

is because the main loop ensures that all states are eventually expanded using either *expand* alone or *expand* followed by *re-expand*. Procedure *expand* uses a stubborn set satisfying the assumptions due to the use of algorithm 2. If *re-expand* is called, it ensures assumption 2b, and assumption 3 remains valid from the call of *expand*.

Regarding termination, the algorithm contains two potentially unbounded loops: the **while**-loop in the main program and the recursion within *traverse*. If the system has a finite number $|\mathbb{W}|$ of states, then the **while**-loop is executed at most $|\underline{\mathbb{W}}| \leq |\mathbb{W}|$ times. Also *traverse* is called at most $|\underline{\mathbb{W}}|$ times, because it is applied only to unexpanded states s and it expands s. So the algorithm is guaranteed to terminate if the system has a finite number of states. If $\mathbb{W}$ is infinite, then the algorithm may terminate but is not guaranteed to terminate. In all cases, if the algorithm terminates, then it produces a finite $\underline{\mathbb{W}}$, ensuring assumption 1 of theorem 2.

We now demonstrate that if algorithm 3 terminates, the reduced state space it produces satisfies assumption 4 of theorem 2. The assumption says that all elementary cycles of the reduced state space contain a state s such that $vis(\Phi) \subseteq TS(s)$. To simplify talking about the assumption, we say that state s is *green* if $vis(\Phi) \subseteq TS(s)$, or if s has no enabled transitions, or if $TS(s)$ contains all enabled transitions; otherwise s is *red*. Assumption 4 can now be rewritten as saying that when algorithm 3 has terminated, $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$ contains no *red* elementary cycles, i.e., elementary cycles $s_0 s_1 \ldots s_n$ such that all of $s_0, s_1, \ldots, s_n$ are red. To see that the new form of assumption 4 is correct, we have to investigate states s such that s has no enabled transitions or $TS(s)$ contains all enabled transitions. If s has no enabled transitions, then it cannot be any of $s_0, s_1, \ldots, s_n$ by the definition of elementary cycles, so its color is not important. If $TS(s)$ contains all enabled transitions, then s has the same output edges and immediate successor states as it would have if $TS(s)$ were $\mathbb{T}$. In other words, in this case we may assign $TS(s) := \mathbb{T}$ immediately before the computation of the new values of $\underline{\mathbb{W}}$ and $\underline{\mathbb{E}}$ in *expand* without modifying the resulting $\underline{\mathbb{W}}$ and $\underline{\mathbb{E}}$. This assignment ensures that $vis(\Phi) \subseteq TS(s)$ and thus justifies the green color of s, without having any other effect. Instead of performing the assignment, *expand* just tests whether $TS(s)$ contains all enabled transitions and modifies the color of s accordingly, because this method has the same effect and is computationally cheaper.

The enabled transitions of $TS(s)$ and the color of s are determined by procedures *expand* and *re-expand*. Procedure *expand*(s) assigns the color of s exactly according to the new formulation of assumption 4. Procedure *re-expand* marks s green and this is correct, because *re-expand* ensures that $T_u \subseteq TS(s)$, and $vis(\Phi) \subseteq T_u$ by the construction of T_u.

Procedure *traverse* is essentially a depth-first search that operates on $(\underline{\mathbb{W}}, \underline{\mathbb{E}}', \mathbb{T})$, where $\underline{\mathbb{E}}'$ is $\underline{\mathbb{E}}$ with output edges of green states removed. The elementary cycles of $(\underline{\mathbb{W}}, \underline{\mathbb{E}}', \mathbb{T})$ are thus exactly the same as the red elementary cycles of $(\underline{\mathbb{W}}, \underline{\mathbb{E}}, \mathbb{T})$. A depth-first search detects an elementary cycle of a graph whenever it encounters a state that is in its search stack. Furthermore, if this never happens, then the

graph contains no elementary cycles. Procedure *traverse*(s) contains this kind of a test for detecting elementary cycles in $(\underline{\mathbb{W}}, \underline{\mathbb{E}}', \mathbb{T})$, where $\underline{\mathbb{E}}'$ is determined by the colors of states valid at the time. The test is indicated by the comment "red cycle test." Whenever such a cycle is encountered, *traverse* "paints" s green by calling *re-expand*(s). As a consequence, the elementary cycle in $(\underline{\mathbb{W}}, \underline{\mathbb{E}}', \mathbb{T})$ disappears immediately after it is detected. Furthermore, *traverse* does not proceed along the output edges of s. This is legal because $(\underline{\mathbb{W}}, \underline{\mathbb{E}}', \mathbb{T})$ does not contain them any more. When *traverse* terminates, the part of $(\underline{\mathbb{W}}, \underline{\mathbb{E}}', \mathbb{T})$ investigated by it contains no elementary cycles. The main loop of algorithm 3 calls *traverse* repeatedly until *traverse* has completely investigated $(\underline{\mathbb{W}}, \underline{\mathbb{E}}', \mathbb{T})$. Therefore, when algorithm 3 terminates, assumption 4 of theorem 2 holds.

Algorithm 3 is more complicated than ordinary state-space generation. First, a typical state-space generation algorithm uses $O(c|\mathbb{T}|)$ time at each state to detect enabled transitions. Factor c arises from the fact that to test one transition it is usually necessary to check the values of the variables referred to by it. Algorithm 3 uses $O(c^2|\mathbb{T}|)$ time, because procedure *expand* involves the execution of algorithm 2. Second, algorithm 3 maintains *search-stack* and the colors of vertices; it also contains some tests that are not needed in ordinary state-space generation. However, with reasonable implementations using customary data structures, none of these is more expensive than $O(c^2|\mathbb{T}|)$ for each state. We conclude that algorithm 3 uses typically only $O(c)$ more time for each state than ordinary state-space generation. The reader should notice, however, that this analysis was based on general assumptions about a "typical" state-space generation algorithm and is not valid in all cases. For instance, it is sometimes possible to find all enabled transitions in a multiprocess program in time $O(c|\mathbb{P}|)$, where $|\mathbb{P}|$ is the number of processes rather than the (usually significantly bigger) number of transitions.

4. Example

To demonstrate the power of the LTL-preserving stubborn set method, we consider a version of the resource-allocator system specified in [3]. Our system consists of a resource allocator and $n \geq 2$ customers that communicate via $2n$ Boolean variables r_i and g_i, initially **F**. The behavior of customer i is shown below.

1: $r_i :=$ **T**; **goto** 2	$/* t_{i1} */$
2: **when** $g_i \Rightarrow$ **goto** 3	$/* t_{i2} */$
3: $r_i :=$**F**; **goto** 4	$/* t_{i3} */$
4: **when** $\neg g_i \Rightarrow$**goto** 1	$/* t_{i4} */$

The customer may use the resource when it is in state 3. The resource allocator behaves as follows:

1: **when** $r_i \Rightarrow g_i := \mathbf{T}$; **goto** $2i$ $\quad /* d_{i1} */$
$2i$: **when** $\neg r_i \Rightarrow g_i := \mathbf{F}$; **goto** 1 $\quad /* d_{i2} */$

The system works basically so that the ith customer may request the resource by moving from 1 to 2, the resource is granted to it when the allocator moves from 1 to $2i$, the customer takes the resource by moving to 3 and releases it by moving to 4, and finally both the allocator and the customer move back to their initial states. The customers may proceed concurrently, but two customers cannot be simultaneously in state 3.

Let us compute the size of the ordinary state space of the system. The states of the variables r_i and g_i can be determined from the states of the customers and the allocator, so they need not be considered. When the resource allocator is in state 1, the customers may be in state 1, 2, or 4 independent of each other and the allocator. This amounts to 3^n states. When the allocator is in state $2i$, the ith customer is in state 2, 3, or 4 and the others are in state 1, 2, or 4, corresponding to $n3^n$ states. This makes a total of $(n + 1)3^n$ states.

4.1. Example system as a v/t system

The resource-allocator system can be seen as a variable/transition system:

$$\mathbb{V} = \{A, C_1, \ldots, C_n, r_1, \ldots, r_n, g_1, \ldots, g_n\}$$

A is the state of the allocator
C_i is the state of the ith customer

$$\mathbb{T} = \{t_{11}, t_{12}, t_{13}, t_{14}, t_{21}, \ldots, t_{24}, \ldots, t_{n1}, \ldots, t_{n4}, d_{11}, d_{12}, d_{21}, d_{22}, \ldots, d_{n1}, d_{n2}\}$$

$$type(A) = \{1, 21, 22, \ldots, 2n\}$$
$$type(C_i) = \{1, 2, 3, 4\}$$
$$type(r_i) = type(g_i) = \{\mathbf{F}, \mathbf{T}\}$$

$$s_0(A) = 1 \wedge \forall i : s_0(C_i) = 1 \wedge s_0(r_i) = s_0(g_i) = \mathbf{F}$$

next is too large to be listed here in full. It can be determined from the program code. For instance, $next(s, t_{12})$ is defined when $s(C_1) = 2 \wedge s(g_1) = \mathbf{T}$. If $next(s, t_{12}) = s'$ (i.e. $s - t_{12} \rightarrow s'$), then $s'(C_1) = 3 \wedge \forall v \neq C_1 : s'(v) = s(v)$.

We need an upper approximation to the set $\{(t, t') \in \mathbb{T} \times \mathbb{T} \mid \neg(t \leftrightarrow t')\}$. If $ref(t) \cap ref(t') = \emptyset$, then $t \leftrightarrow t'$, because then the occurrence of t does not directly affect the environment of t' and vice versa. We conclude that the union of $ref(v)^2$ for every variable v of the system is an upper approximation to the set. We continue by investigating how the approximation can be improved.

We have $ref(A) = \{d_{11}, d_{12}, d_{21}, d_{22}, \ldots, d_{n1}, d_{n2}\}$. Because of the control structure of the resource allocator, transitions of the form d_{i2} are never enabled simultaneously with any other transitions in $ref(A)$. Thus the left-hand side of the implication in the definition of "$\leftrightarrow$" is never satisfied, and the implication is always true, if $t = d_{i2}$ and $t' \in ref(A) - \{d_{i2}\}$. Consequently, the corresponding transition pairs (t, t') and (t', t) can be eliminated. A transition seldom accords with itself, so we choose not to try to eliminate the pairs (d_{i2}, d_{i2}). Because d_{i1} can disable d_{j1}, $\neg(d_{i1} \leftrightarrow d_{j1})$ holds when $1 \leq i \leq n$ and $1 \leq j \leq n$. As a result, the pairs (d_{i1}, d_{j1}) remain. So we have eliminated all pairs except (d_{i2}, d_{i2}) and (d_{i1}, d_{j1}), where $1 \leq i \leq n$ and $1 \leq j \leq n$. By similar argument all pairs except $\{(t, t) \mid t \in ref(C_i)\}$ can be eliminated from the sets $ref(C_i)^2$, because $ref(C_i) = \{t_{i1}, t_{i2}, t_{i3}, t_{i4}\}$.

Since $ref(r_i) = \{t_{i1}, t_{i3}, d_{i1}, d_{i2}\}$, we have investigated $ref(r_i)^2$ except the pairs (t_{ij}, d_{ik}) and (d_{ik}, t_{ij}), where $j \in \{1, 3\}$ and $k \in \{1, 2\}$. Consider the states s enabling both t_{i1} and d_{i1}. We have $s(r_i) = \mathbf{T}$ and $s(C_i) = s(A) = 1$. When t_{i1} or d_{i1} occurs, the only variables whose values may be changed are r_i, g_i, C_i, and A, so we investigate them only. If $s \xrightarrow{-t_{i1}\rightarrow} s_1$ and $s \xrightarrow{-d_{i1}\rightarrow} s'$, then $s_1(r_i) = s'(r_i) = \mathbf{T}$, $s_1(g_i) = s(g_i)$, $s'(g_i) = \mathbf{T}$, $s_1(C_i) = 2$, $s'(C_i) = 1 = s_1(A)$, and $s'(A) = 2i$. Therefore, $en(s_1, d_{i1})$ and $en(s', t_{i1})$. Let $s_1 \xrightarrow{-d_{i1}\rightarrow} s_1'$ and $s' \xrightarrow{-t_{i1}\rightarrow} s_1''$. By computing the values of r_i, g_i, C_i, and A in s_1' and s_1'', we see that $s_1' = s_1''$. Thus $t_{i1} \leftrightarrow d_{i1}$. By similar argument it can be shown that $t_{i3} \leftrightarrow d_{i2}$. Only the pairs (t_{i1}, d_{i2}), (t_{i3}, d_{i1}) and their inverses and the pairs of the form (t, t) were not eliminated from $ref(r_i)^2$. Investigating $ref(g_i)$ in the similar way leaves as the total only the following pairs and their inverses:

$$(t, t), (d_{i1}, d_{j1}), (t_{i1}, d_{i2}), (t_{i2}, d_{i2}), (t_{i3}, d_{i1}), (t_{i4}, d_{i1}),$$
$$\text{where } 1 \leq i \leq n, 1 \leq j \leq n \text{ and } t \in \mathbb{T}.$$

Also the predicates $en(s, t, V)$ and the sets $wrup(t, V)$ for $t \in \mathbb{T}$ and for some $V \subseteq \mathbb{V}$ are needed by the stubborn set method. Consider t_{12}, for example. We have $en(s, t_{12}) \Leftrightarrow s(C_1) = 2 \wedge s(g_1) = \mathbf{T}$. Therefore $en(s, t_{12}, \{C_1\}) \Leftrightarrow s(C_1) = 2$ and $en(s, t_{12}, \{g_1\}) \Leftrightarrow s(g_1) = \mathbf{T}$. We may choose $wrup(t_{12}, \{C_1\}) = \{t_{11}\}$, because t_{11} is the only transition whose occurrence can assign 2 to C_1. Similarly $wrup(t_{12}, \{g_1\}) = \{d_{11}\}$. Following the same principle, we can evaluate $en(s, t, \{v\})$ and define $wrup(t, \{v\})$ for every $t \in \mathbb{T}$ and $v \in ref(t)$. All the so-defined *wrup* sets contain exactly one transition, excluding the sets $wrup(d_{i1}, \{A\})$, which are all equal to $\{d_{12}, d_{22}, \ldots, d_{n2}\}$.

4.2. Reduced state space of the example system

Now we construct a reduced state space of the resource-allocator system. We want to know whether the system guarantees that the resource cannot be used simultaneously by two customers (it does), and whether a customer that has requested a resource eventually uses the resource (not true). Because of the

symmetry of the system, the first requirement can be specified by the LTL formula $\Box((C_1 \neq 3) \vee (C_2 \neq 3))$, where C_1 and C_2 are the states of customers 1 and 2, respectively. The second requirement can be encoded as $\Box((C_1 = 2) \Rightarrow \Diamond(C_1 = 3))$. The transitions that can modify the truth values of the state predicates $C_1 \neq 3 \vee C_2 \neq 3$, $C_1 = 2$ and $C_1 = 3$ are $t_{11}, t_{12}, t_{13}, t_{22}$, and t_{23}. Thus we choose $vis(\Phi) = \{t_{11}, t_{12}, t_{13}, t_{22}, t_{23}\}$.

In the initial state s_0, the transitions t_{i1} and no other transitions are enabled. If we want to build a stubborn set $TS(s_0)$ around t_{i1}, then we have to take d_{i2} into the set because $\neg(t_{i1} \leftrightarrow d_{i2})$. Transition d_{i2} is disabled; thus we have to find $V \subseteq \mathbb{V}$ such that $\neg en(s_0, d_{i2}, V)$ and include $wrup(d_{i2}, V)$ to the set. We can choose $V = \{A\}$ and $wrup(d_{i2}, \{A\}) = \{d_{i1}\}$. We have $\neg en(s_0, d_{i1}, \{r_i\})$ and $wrup(d_{i1}, \{r_i\}) = \{t_{i1}\}$. Because t_{i1} is already in the set, we can stop with the set $TS(s_0) = \{t_{i1}, d_{i1}, d_{i2}\}$. It is stubborn and contains exactly one enabled transition, namely, t_{i1}. To satisfy assumption 3 of theorem 2, it is reasonable to choose the value of i so that $i > 1$, and step 1 of algorithm 2 indeed does so. The exact value of i depends on implementation details of algorithm 2, but fortunately we do not need it to continue our analysis. In conclusion, only one transition is investigated in s_0, namely, t_{i1}. Let us call the resulting new state s'.

By applying the above reasoning again, one can see that if $j \neq i, \{t_{j1}, d_{j1}, d_{j2}\}$ is stubborn in s'. In an attempt to avoid visible transitions, step 1 of algorithm 2 chooses $\{t_{j1}, d_{j1}, d_{j2}\}$ for some $j \geq 2$ and so on, until the state $s_{12\cdots}$ such that $s_{12\cdots}(A) = s_{12\cdots}(C_1) = 1$ and $s_{12\cdots}(C_i) = 2$ for $2 \leq i \leq n$ is reached. At this stage we have generated n states.

The enabled transitions at $s_{12\cdots}$ are t_{11} and d_{i1} for $i \geq 2$. Assumption 3 forces us to include at least one of d_{i1} into $TS(s_{12\cdots})$. Because $\neg(d_{i1} \leftrightarrow d_{j1})$ we conclude that $\forall i \geq 1 : d_{i1} \in TS(s_{12\cdots})$. Intuitively, this reflects the fact that it is essential which customer gets the resource. Transition d_{11} is disabled and, as before, takes us to t_{11}. That is, C_1 is also given the chance to take the resource. So we have to take all enabled transitions into $TS(s_{12\cdots})$. Therefore, algorithm 2 also takes all enabled transitions and algorithm 3 marks $s_{12\cdots}$ green. In essence, the LTL-preserving stubborn set method gives all the other customers the chance to take the resource before t_{11} occurs, because the occurrence of t_{11} modifies the value of the state predicate $C_1 = 2$ in Φ. The method tries to find out what can happen before the state predicate value is modified.

Let $d_{i1}, i > 2$, occur in $s_{12\cdots}$. The system moves to a state where $C_i = 2$ and $A = 2i$. As before, the stubborn set method concentrates on the invisible transitions and executes $t_{i2}, t_{i3}, d_{i2}, t_{i4}$, and t_{i1} returning to $s_{12\cdots}$. For exactly one i, the state immediately before the occurrence of t_{i1} has already been generated. So we have generated a total of $5(n-2)-1$ new states. (Strictly speaking, the already generated state may correspond to $i = 2$, but this case is similar and treated below, so there is no net error in the state count.)

Now consider the occurrence of t_{11} in $s_{12\cdots}$ resulting in the state $s_{22\cdots}$, where $\forall i : s_{22\cdots}(C_i) = 2 \wedge s_{22\cdots}(A) = 1$. Transitions $d_{i1}, i \geq 1$, are enabled. They do not accord with each other, so we have to consider all of them. Algorithm 3 paints

$s_{22^\bullet}$ green. After the occurrence of $d_{i1}, i > 2$, we continue as above by firing $t_{i2}, t_{i3}, d_{i2}, t_{i4}$, and t_{i1} in sequence and return to $s_{22^\bullet}$. With $i \leq 2$ we have the problem that $t_{11}, t_{12}, t_{13}, t_{22}$, and t_{23} are visible; thus the above reasoning does not immediately apply. But it turns out that when t_{12}, t_{13}, t_{22}, or t_{23} occurs, it is the only enabled transition, and therefore there is no problem. When t_{11} is due to occur, the state proves to be $s_{12^\bullet}$, which has already been investigated. We have generated $5n$ new states.

We still have not investigated the occurrence of d_{21} in $s_{12^\bullet}$. The resulting pattern resembles the case of $d_{i1}, i > 2$, occurring in $s_{12^\bullet}$, but the fact that t_{22} and t_{23} are visible has to be taken into account. Its effect is that also t_{11} has to be investigated when t_{22} or t_{23} occurs. But the two states resulting from the occurrences of t_{11} have already been generated as indirect successors of $s_{22^\bullet}$. Thus five new states emerge.

The total number of states generated is thus $11n - 6$. Every elementary cycle of the reduced state space contains either $s_{12^\bullet}$ or $s_{22^\bullet}$ which are both green; thus assumption 4 of theorem 2 is satisfied without further action. The validity of the formulas $\Box((C_1 \neq 3) \vee (C_2 \neq 3))$ and $\Box((C_1 = 2) \Rightarrow \Diamond(C_1 = 3))$ can now be checked from the reduced state space. The former is true; the latter is not. The reduced state space is linear in the number of customers, while the ordinary state space is exponential.

5. Concluding remarks

We showed how to generate reduced state spaces such that the truth values of LTL formulas are preserved, provided that the formulas are given before the reduced state-space generation commences and that they do not contain the "next state" operator. In the example in section 4, the reduction of the size of the state space is from exponential to linear in the size of the system. This is a very good result. However, it is currently not known how well the LTL-preserving stubborn set method performs on the average. One may expect that the size of the reduced state space increases when more and more variables are referred to by the formulas to be preserved.

At this writing, the LTL-preserving stubborn set method has not been implemented. However, related stubborn set state-space reduction methods have been implemented into two tools: *Toras* [22] and *ARA* [23]. Toras is being developed in Telecom Australia Research Laboratories. Among other features, it supports a version of the stubborn set method that preserves the *failure semantics* [24] of systems. The method differs from the one presented in this article in that it does not need assumption 4 of theorem 2. To give an example of the performance of Toras, a certain version of the n dining philosophers system has $3^n - 1$ states, and the deadlock-preserving stubborn set method reduces the number to $3n^2 - 3n + 2$ states. For the 100-philosopher system ($\approx 10^{47}$states), Toras generated the predicted 29,702 states in 20 minutes CPU time on a Sun^{TM} 3/60

workstation [22].

The tool ARA is being developed in the Computer Technology Laboratory of the Technical Research Centre of Finland. It invokes a stubborn set method preserving the *CFFD semantics* [25] of systems. The method is essentially the same as the LTL-preserving stubborn set method of this article, but the implementation uses a process algebraic context, which is quite different from v/t systems and ill suited for the analysis of temporal logic formulas. The CFFD-preserving stubborn set method transforms the number of states of a token-ring system described originally in [26] from $9n2^{n-2}$ to $5n$, where n is the number of customers in the system. ARA used approximately three minutes on a PC to construct a 125-state reduced state space of the 25-customer token-ring system. The full state space would have contained approximately 1.9×10^9 states.

For the LTL-preserving stubborn set method to be useful, there must be a tool for model checking. The principles of a suitable model-checking algorithm are presented in [2]. Another possibility is to adopt from [15] the idea of representing the formula as a Büchi automaton B connected in parallel with the system. With this approach, assumption 3 of theorem 2 need not be satisfied when B is not in an accepting state. Assumptions 3 and 4 of theorem 2 guarantee that the reduced state space contains an accepting execution of B if and only if the full state space does, so the assumption in [15] of "nonblocking processes" is not needed. This approach gives the LTL-preserving stubborn set method the benefits of "on-the-fly" verification discussed in the introduction. However, the modeling of fairness constraints without adopting the nonblocking processes assumption remains an open question.

Acknowledgments

This work has been supported by the Technology Development Centre of Finland (TEKES). The anonymous referees for this article gave useful comments.

References

[1] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8 (2): 244–263, 1986.

[2] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. Proceedings of the Twelfth ACM Symposium on the Principles of Programming Languages, New Orleans, LA, January 1985, pp. 97–107.

[3] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency, Lecture Notes in Computer Science*, 224: 510–584, 1986.

[4] A. Valmari.*State Space Generation: Efficiency and Practicality.* Ph.D. thesis, Tampere University of Technology Publications 55, Tampere, Finland, 1988.

[5] W.T. Overman. *Verification of Concurrent Systems: Function and Timing.* Ph.D. dissertation,

University of California Los Angeles, 1981.

[6] A. Valmari. Error detection by reduced reachability graph generation. Proceedings of the Ninth European Workshop on Application and Theory of Petri Nets, Venice, Italy, pp. 95–112, 1988.

[7] A. Valmari. Heuristics for lazy state generation speeds up analysis of concurrent systems. Proceedings of the Finnish Artificial Intelligence Symposium STeP-88, Helsinki, Vol. 2, pp. 640–650, 1988.

[8] A. Valmari. Eliminating redundant interleavings during concurrent program verification. Proceedings of Parallel Architectures and Languages Europe '89, Eindhoven, The Netherlands, June 1989, Vol. 2. *Lecture Notes in Computer Science*, 366: 89–103, 1989.

[9] A. Valmari. Stubborn sets for reduced state space generation. *Advances in Petri Nets 1990. Lecture Notes in Computer Science*, 483: 491–515, 1991. (An earlier version appeared in Proceedings of the Tenth International Conference on Application and Theory of Petri Nets, Bonn, FRG, Vol. II, pp. 1–22, 1989.)

[10] A. Valmari. Stubborn sets of coloured Petri nets. Proceedings of the 12th International Conference on Application and Theory of Petri Nets, Gjern, Denmark, pp. 102–121, 1991.

[11] A. Valmari and M. Clegg. Reduced labelled transition systems save verification effort. *Proceedings of CONCUR '91*, Amsterdam, The Netherlands, August 1991. *Lecture Notes in Computer Science*, 527: 526–540, 1991.

[12] A. Valmari. A stubborn attack on state explosion. Computer-Aided Verification '90, New Brunswick, NJ (proceedings of a workshop). *AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 3, pp. 25–41. American Mathematical Society, 1991. (An abbreviated version appeared in Computer-Aided Verification, 2nd International Conference, *Lecture Notes in Computer Science*, 531: 156–165, 1991.)

[13] P. Godefroid. Using partial orders to improve automatic verification methods. Computer-Aided Verification '90, New Brunswick, NJ (proceedings of a workshop). *AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 3. American Mathematical Society, 1991, pp. 321–340.

[14] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock-freedom and safety properties. Proceedings of Computer-Aided Verification '91, Aalborg, Denmark, July 1991. *Lecture Notes in Computer Science* 575: 332–342, 1992.

[15] P. Godefroid and P. Wolper. A partial approach to model checking. Proceedings of the 6th Symposium on Logic in Computer Science, Amsterdam, The Netherlands, pp. 406–415, July 1991.

[16] M. Itoh and H. Ichikawa. Protocol verification algorithm using reduced reachability analysis. *Transactions of the IECE of Japan*, E66(2): 88–93, 1983.

[17] R. Janicki and M. Koutny. On some implementation of optimal simulations. Computer-Aided Verification '90, New Brunswick, NJ, (proceedings of a workshop). *AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 3. American Mathematical Society, 1991, pp. 231–250.

[18] J. Quemada. Compressed state space representation in LOTOS with the interleaved expansion. *Protocol Specification, Testing and Verification XI* (Proceedings of the 11th International IFIP WG 6.1 Symposium, Stockholm, Sweden, June 1991). North-Holland, Amsterdam, 1991, pp. 19–35.

[19] S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75: 263–287, 1990.

[20] L. Lamport. What good is temporal logic? Information Processing '83, *Proceedings of the IFIP 9th World Computer Congress*. North-Holland, Amsterdam, pp. 657–668.

[21] A.V. Aho, J.E. Hopcroft and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[22] G.R. Wheeler, A. Valmari, and J. Billington. Baby Toras eats philosophers but thinks about solitaire. Proceedings of the Fifth Australian Software Engineering Conference, Sydney, NSW, Australia 1990, pp. 283–288.

[23] J. Kemppainen, M. Levanto, A. Valmari, and M. Clegg. "ARA" puts advanced reachability analysis methods together. Tampere University of Technology, Software Systems Laboratory Report 14: Proceedings of the Fifth Nordic Workshop on Programming Environment Research, Tampere, Finland, January 1992.

[24] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31 (3): 560–599, 1984.

[25] A. Valmari and M. Tienari. An improved failures equivalence for finite-state systems with a reduction algorithm. *Protocol Specification, Testing and Verification XI* (Proceedings of the 11th International IFIP WG 6.1 Symposium, Stockholm, Sweden, June 1991). North-Holland, Amsterdam, 1991, pp. 3–18.

[26] S. Graf and B. Steffen. Compositional minimization of finite-state processes. Computer-Aided Verification '90, New Brunswick, NJ, (proceedings of a workshop). *AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 3. American Mathematical Society, 1991, pp. 57–73.