A STUBBORN ATTACK ON STATE EXPLOSION (abridged version)

Antti Valmari
Technical Research Centre of Finland
Computer Technology Laboratory
PO Box 201
SF-90571 Oulu
FINLAND

Tel. +358 81 509 111

ABSTRACT

The paper 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 a collection of linear temporal logic formulas are the same in the ordinary and reduced state spaces. The only restrictions posed by the method are that the collection of formulas must be known before the reduced state space generation is commenced, the use of the temporal operator "next" is prohibited, and the (reduced) state space of the system must be finite. The method cuts down the number of states by utilising the fact that in concurrent systems the nett result of the occurrence of two events is often independent of the order of occurrence.*

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 [Clarke & 86] [Lichtenstein & 85]. A well known problem of the approach is that the state spaces of systems tend to be very large, rendering the verification of medium-size and large non-trivial 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 which 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 has been recognized long ago and the choice of a coarser level of atomicity has been suggested as a partial solution (see [Pnueli 86]). Unfortunately, the power of coarsening the level of atomicity is limited. Consider a system consisting of n processes which 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 [Valmari 88c]. 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 [Overman 81]. Overman's work is little known, perhaps because he considered a very restricted case (the terminal states of systems consisting of processes which 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 effectively solved by Valmari when he presented the so-called *stubborn set* method [Valmari 88a, 88b]. The stubborn set method has been developed in a series of papers [Valmari 88a, 88b, 88c, 89a, 89b]. Originally the method could be used only to investigate deadlocks but more advanced versions of the method have been gradually developed in order to verify more properties. The method was initially applied to ordinary Petri nets but now it is applicable to a rather general model of concurrency, the *variable/transition* systems. Two profoundly different versions of the method have been distinguished: *weak* and *strong*. The weak theory is more complicated and more difficult to implement, but it leads to better reduction results.

The present paper extends the stubborn set method to almost full linear temporal logic — almost, because the operator "next state" is forbidden. 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. The theory is developed in Chapter 2. Chapter 3 (not in this abridged version) discusses how the theory may be implemented and Chapter 4 contains an example.

This paper is a revised and abridged version of a paper with the same name which appeared in DIMACS Technical Report 90-31, "Workshop on Computer-Aided Verification", Rutgers University, NJ, USA, June 1990, Volume I.

2. DEFINITIONS AND BASIC THEOREMS

2.1 Variable/Transition Systems

To develop the stubborn set method we look at concurrent systems as systems 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 $\mathbb{S}$ which 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 $s -t\rightarrow s'$. We often merge the states between successive transition occurrences and write $s_0 -t_1\rightarrow s_1 -t_2\rightarrow \ldots -t_n\rightarrow s_n$ instead of $s_0 -t_1\rightarrow s_1 \wedge s_1 -t_2\rightarrow s_2 \wedge \ldots \wedge s_{n-1} -t_n\rightarrow s_n$. A sequence like this is called a *finite execution sequence* and its *length* is n. The *concatenation* of the execution sequences $\sigma = s_0 -t_1\rightarrow \ldots -t_n\rightarrow s_n$ and $\rho = r_0 -d_1\rightarrow \ldots -d_m\rightarrow r_m$ where $s_n = r_0$ is defined by $\sigma^\circ\rho = s_0 -t_1\rightarrow \ldots -t_n\rightarrow s_n -d_1\rightarrow \ldots -d_m\rightarrow r_m$. "$\rightarrow$" is defined by $s \rightarrow s' \Leftrightarrow \exists\, t \in \mathbb{T}: s -t\rightarrow s'$. "$\rightarrow^*$" is the reflexive transitive closure of "$\rightarrow$". There is a distinguished state $\mathbf{s}_0$ called the *initial state* of the system. A *variable/transition system* or *v/t-system* is the 5-tuple $(\mathbb{V},\mathbb{T},type,next,\mathbf{s}_0)$, where the components of the 5-tuple are as just explained.

Section 4.1 contains a non-trivial 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 2.1 Let $t, t' \in \mathbf{T}$, $s \in \mathbf{S}$, $V \subseteq \mathbf{V}$ and $T \subseteq \mathbf{T}$.

- t is *enabled with respect to* V at s, denoted by $en(s,t,V)$, iff
  $\exists\, s' \in \mathbf{S}: en(s',t) \wedge \forall\, v \in V: s'(v) = s(v)$.
- T is a *write up* set of t with respect to V, iff for every $t' \in \mathbf{T}$ and $s' \in \mathbf{S}$
  $\neg\, en(s,t,V) \wedge s \xrightarrow{t'} s' \wedge en(s',t,V) \Rightarrow t' \in T$.
- t *accords with* t', denoted by $t \leftrightarrow t'$, iff for every $s \in \mathbf{S}$
  $en(s,t) \wedge en(s,t') \Rightarrow \exists\, s', s_1, s'_1 \in \mathbf{S}: s \xrightarrow{t} s_1 \xrightarrow{t'} s'_1 \wedge s \xrightarrow{t'} s' \xrightarrow{t} s'_1$. □

The intuition behind the definition of $en(s,t,V)$ is perhaps best understood by noticing that if t is *not* enabled w.r.t. 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: $en(s,t) \Leftrightarrow en(s,t,\mathbf{V}) \Leftrightarrow \forall\, V \subseteq \mathbf{V}: en(s,t,V)$.

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 which have the potential of modifying the status of t from disabled w.r.t. V to enabled w.r.t. 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 w.r.t. 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 which 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 $\mathbf{T}$. From now on we assume that a unique write up set is defined for every (t,V)-pair. We denote it by $wrup(t,V)$.

The definition of according with can be illustrated graphically:

$$\begin{tikzcd} s \arrow[r, "t'"] \arrow[d, "t"] & s' & & s \arrow[r, "t'"] \arrow[d, "t"] & s' \arrow[d, "t"] \\ s_1 & & \Rightarrow & s_1 \arrow[r, "t'"] & s'_1 \end{tikzcd}$$

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 call the set of variables tested, read or written by a transition its *reference set*. Two transitions accord with each other if their 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 independent of to what variables they refer, because they are never simultaneously enabled.

2.2 Stubborn Sets

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

Definition 2.2 Let $T \subseteq \mathbf{T}$ and $s \in \mathbf{S}$. T is *semistubborn* at s, iff $\forall\, t \in T$:

(1) $\neg\, en(s,t) \Rightarrow \exists\, V \subseteq \mathbf{V}: \neg\, en(s,t,V) \wedge wrup(t,V) \subseteq T$

(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 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 **T** are semistubborn at every state, 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:

Theorem 2.3 Let $T \subseteq \mathbf{T}$ and $s_0 \in \mathbf{S}$ such that T is semistubborn at s_0. Let $n \geq 1$ and

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

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. σ corresponds to the top and right edges of the figure, while σ' corresponds to the left and bottom sides.

$$\begin{tikzcd}
s_0 \arrow[r, "-t_1\rightarrow", phantom] \arrow[d, "t_n"] & s_1 \arrow[r, "-t_2\rightarrow", phantom] & \ldots \arrow[r, "-t_{n-2}\rightarrow", phantom] & s_{n-2} \arrow[r, "-t_{n-1}\rightarrow", phantom] & s_{n-1} \arrow[d, "t_n"] \\
s'_0 \arrow[r, "-t_1\rightarrow", phantom] & s'_1 \arrow[r, "-t_2\rightarrow", phantom] & \ldots \arrow[r, "-t_{n-2}\rightarrow", phantom] & s'_{n-2} \arrow[r, "-t_{n-1}\rightarrow", phantom] & s'_{n-1} \\
 & & & & = s_n
\end{tikzcd}$$

The proof is a straightforward application of the definitions, and can be found in [Valmari 88b, 89a, 89b].

Theorem 2.3 allows us to move the occurrence of a transition belonging to a semistubborn set from 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 2.4 Let $T \subseteq \mathbf{T}$ and $s \in \mathbf{S}$. 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 **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.

2.3 Reduced State Spaces

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

Definition 2.5 The *state space* of the *v/t-system* $(\mathbf{V},\mathbf{T},type,next,\mathbf{s}_0)$ is the triple $(\mathbf{W},\mathbf{E},\mathbf{T})$, where

- $\mathbf{W} = \{s \in \mathbf{S} \mid \mathbf{s}_0 \rightarrow^* s\}$
- $\mathbf{E} = \{(s,t,s') \in \mathbf{W} \times \mathbf{T} \times \mathbf{W} \mid s -t\rightarrow s'\}$ □

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 $\mathbf{s}_0$. Assume 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:

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

Definition 2.6 Assume a stubborn set generator TS is given. The *reduced state space* of the v/t-system $(\mathbb{V},\mathbb{T},type,next,\mathbf{s}_0)$ is the triple $(\underline{\mathbb{W}},\underline{\mathbb{E}},\mathbb{T})$, where

- $\underline{\mathbb{W}} = \{s \in \mathbb{S} \mid \mathbf{s}_0 \underline{\rightarrow}^* s\}$
- $\underline{\mathbb{E}} = \{(s,t,s') \in \underline{\mathbb{W}} \times \mathbb{T} \times \underline{\mathbb{W}} \mid s \underline{-t\rightarrow} s'\}$ □

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 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 [Valmari 88c]. 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. [Valmari 88a, 88b] give a linear and an (under certain reasonable assumptions) quadratic algorithm for finding almost optimal and optimal stubborn sets, respectively. 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 optimises it as much as it can and stops without investigating the rest of the v/t-system. The linear algorithm is briefly described in Chapter 3 of the unabridged version.

2.4 Execution Sequences in Reduced State Spaces

This section is devoted to a construction which, given a finite execution sequence of the system under analysis, finds an execution sequence which 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. σ is the original finite execution sequence. 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. $k(i)$ is 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 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 2.7 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. $k(i)$ 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, ρ_i consists of one state and no transition occurrences, the construction cannot be continued.
- If $k(i) > 0$, $d_{1,i}$ is enabled at $r_{0,i}$, 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 2.4 $TS(r_{0,i})$ contains an enabled transition t'_i. By (2) of Definition 2.2 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 $1 \le j(i) \le k'(i)$ 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 2.3 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 $\sigma_{i+1} = \sigma'_i$. ρ_{i+1} is defined by $\rho''_i = r_{0,i} -d_{j(i),i}\rightarrow \rho_{i+1}$. After this i:th 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 $d_{j(i),i}$ has moved to a different location, or a new transition occurrence 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 which are finite and 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 $vis(\Phi)$ 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'. $vis(\Phi)$ is a set containing at least the properly visible transitions. The reason why we allow in $vis(\Phi)$ the presence of transitions which are not properly visible is the same as the reason of allowing the write up set to be an upwards approximation of the smallest set with the required property. Transition t is *visible* if $t \in vis(\Phi)$, 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 of 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 which 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 [Lamport 83]).

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

Theorem 2.8 Let $(\mathbf{V},\mathbf{T},type,next,\mathbf{s}_0)$ be a v/t-system and $\mathbf{S}$ and $(\mathbf{W},\mathbf{E},\mathbf{T})$ 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 $\mathbf{S}$. Let TS be a stubborn set generator and $(\underline{\mathbf{W}},\underline{\mathbf{E}},\underline{\mathbf{T}})$ the corresponding reduced state space such that the following hold:

(1) $\underline{\mathbf{W}}$ is finite

(2) For every $\underline{s} \in \underline{\mathbf{W}}$, either
  (a) $TS(\underline{s})$ contains no enabled visible transitions, or
  (b) $vis(\Phi) \subseteq TS(\underline{s})$

(3) For every $\underline{s} \in \underline{\mathbf{W}}$, if there is an enabled invisible transition, then $TS(\underline{s})$ contains an enabled invisible transition

(4) Every elementary cycle of $(\underline{\mathbf{W}},\underline{\mathbf{E}},\underline{\mathbf{T}})$ contains at least one state $\underline{s}$ such that $vis(\Phi) \subseteq TS(\underline{s})$.

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

(For proof see the unabridged version of this paper.)

3. IMPLEMENTATION OF THE METHOD

(This abridged version does not contain Chapter 3.)

4. EXAMPLE

To demonstrate the power of the LTL preserving stubborn set method we consider a version of the resource allocator system specified in [Pnueli 86]. Our system consists of a resource allocator and $n \geq 2$ customers which communicate via $2n$ Boolean variables r_i and g_i, initially F. The behaviour of customer i is shown below.

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

The system has $(n+1)\cdot 3^n$ states (for proof see the unabridged version).

4.1 Example System as a V/T-System

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

$\mathbf{V} = \{A, C_1, \ldots, C_n, r_1, \ldots, r_n, g_1, \ldots, g_n\}$
A is the state of the allocator and C_i is the state of the i:th customer

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

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

$\mathbf{s}_0(A) = 1 \wedge \forall\, i\colon \mathbf{s}_0(C_i) = 1 \wedge \mathbf{s}_0(r_i) = \mathbf{s}_0(g_i) = \mathbf{F}$

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 if $s(C_1) = 2 \wedge s(g_1) = \mathbf{T}$. If $next(s,t_{12}) = s'$ (i.e. $s \xrightarrow{t_{12}} s'$) then $s'(C_1) = 3 \wedge \forall\, v \neq C_1\colon s'(v) = s(v)$.

We need an upper approximation to the set $\{(t,t') \in \mathbb{T}\times\mathbb{T} \mid \neg\, t \leftrightarrow t'\}$. If the sets of variables referred to by t and t' are disjoint, then $t \leftrightarrow t'$, because then the occurrence of t does not directly affect the environment of t' and vice versa. Let ref(v) denote the set of transitions referring to a variable v. 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 transition 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}). $\neg\, d_{i1} \leftrightarrow d_{j1}$ holds for $1 \leq i,j \leq n$, because d_{i1} can disable d_{j1}. 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 $1 \leq i,j \leq n$. By similar argument all pairs except $\{(t,t) \mid t \in \text{ref}(C_i)\}$ can be eliminated from the sets ref$(C_i)^2$, as ref$(C_i) = \{t_{i1}, t_{i2}, t_{i3}, t_{i4}\}$.

As 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 $s \xrightarrow{t_{i1}} s_1$ and $s \xrightarrow{d_{i1}} s'$ 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 $s_1 \xrightarrow{d_{i1}} s'_1$ and $s' \xrightarrow{t_{i1}} s''_1$. 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 left:

(t,t), (d_{i1}, d_{j1}), (t_{i1}, d_{i2}), (t_{i2}, d_{i2}), (t_{i3}, d_{i1}), (t_{i4}, d_{i1}), where $1 \leq i,j \leq n$ 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 $V \subseteq \mathbf{V}$ are needed by the stubborn set method. Consider t_{12}, for example. $en(s,t_{12})$ holds iff $s(C_1) = 2 \wedge s(g_1) = \mathbf{T}$. 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 it is the only transition whose occurrence can assign 2 to C_1. Similarly $wrup(t_{12},\{g_1\}) = \{d_{11}\}$. Following the same principles we can evaluate $en(s,t,\{v\})$ and define $wrup(t,\{v\})$ for every $t \in \mathbb{T}$ and $v \in \mathbf{V}$. 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 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 which 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 which 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 $\mathbf{s}_0$ the transitions t_{i1} and no other transitions are enabled. If we want to build a stubborn set $TS(\mathbf{s}_0)$ around t_{i1} then we have to take d_{i2} into the set because $\neg\, t_{i1} \leftrightarrow d_{i2}$. d_{i2} is disabled, thus we have to find $V \subseteq \mathbf{V}$ such that $\neg\, en(\mathbf{s}_0, d_{i2}, V)$ and include $wrup(d_{i2}, V)$ to the set. We can choose $V = \{A\}$ and $wrup(d_{i2}, \{A\}) = \{d_{i1}\}$. $\neg\, en(\mathbf{s}_0, d_{i1}, \{r_i\})$ holds and $wrup(d_{i1}, \{r_i\}) = \{t_{i1}\}$, but t_{i1} is already in the set. We can thus stop with the set $TS(\mathbf{s}_0) = \{t_{i1}, d_{i1}, d_{i2}\}$. It is stubborn and contains exactly one enabled transition, namely t_{i1}. To satisfy Assumption (3) of Theorem 2.8 it is reasonable to choose the i so that $i > 1$, and the algorithm in Chapter 3 indeed does so. So only one transition is fired in $\mathbf{s}_0$, namely t_{i1}. 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, the algorithm in Chapter 3 chooses $\{t_{j1}, d_{j1}, d_{j2}\}$ for some $j \geq 2$ and so on until the state s_{12*} such that $s_{12*}(A) = s_{12*}(C_1) = 1$ and $s_{12*}(C_i) = 2$ for $2 \leq i \leq n$ is reached. At this stage we have generated n states.

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

For the continuation of the reduced state space generation see the unabridged version of this paper. Assumption (4) of Theorem 2.8 is satisfied without further action. The total number of states generated is $11n-6$.

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 they do not contain the "next state" operator. In the example in Chapter 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.

By the time of the writing of this paper the LTL preserving stubborn set method has not been implemented. However, a related stubborn set state space reduction method has been implemented into a tool called *Toras* [Wheeler & 90]. Toras is being developed in Telecom Australia Research Laboratories. Among other features, it supports an as yet unpublished version of the stubborn set method which preserves the *failure semantics* [Brookes & 84] of systems. The method differs from the one presented in this paper in that it does not need Assumptions (1) and (4) of Theorem 2.8. 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 basic 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 3/60 [Wheeler & 90].

ACKNOWLEDGEMENTS

This work has been supported by the Technology Development Centre of Finland (TEKES).

REFERENCES

[Aho & 74] Aho, A. V., Hopcroft, J. E. & Ullman, J. D.: *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts 1974, 470 p.

[Brookes & 84] Brookes, S. D., Hoare, C. A. R. & Roscoe, A. W.: *A Theory of Communicating Sequential Processes.* Journal of the ACM 31 (3) 1984, pp. 560–599.

[Clarke & 86] Clarke, E. M., Emerson, E. A. & Sistla, A. P.: *Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications.* ACM Transactions on Programming Languages and Systems 8 (2) 1986 pp. 244–263.

[Lamport 83] Lamport, L.: *What Good is Temporal Logic?* Information Processing '83, North-Holland pp. 657–668.

[Lichtenstein & 85] Lichtenstein, O. & Pnueli, A.: *Checking that Finite State Concurrent Programs Satisfy their Linear Specification.* Proceedings of the Twelfth ACM Symposium on the Principles of Programming Languages, January 1985 pp. 97–107.

[Overman 81] Overman, W. T.: *Verification of Concurrent Systems: Function and Timing.* Ph.D. Dissertation, University of California Los Angeles 1981, 174 p.

[Pnueli 86] Pnueli, A.: *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, Springer 1986 pp. 510–584.

[Valmari 88a] Valmari, A.: *Error Detection by Reduced Reachability Graph Generation.* Proceedings of the Ninth European Workshop on Application and Theory of Petri Nets, Venice, Italy 1988 pp. 95–112.

[Valmari 88b] Valmari, A.: *Heuristics for Lazy State Generation Speeds up Analysis of Concurrent Systems.* Proceedings of the Finnish Artificial Intelligence Symposium STeP-88, Helsinki 1988 Vol. 2 pp. 640–650.

[Valmari 88c] Valmari, A.: *State Space Generation: Efficiency and Practicality.* Ph.D. Thesis, Tampere University of Technology Publications 55, 1988, 169 p.

[Valmari 89a] Valmari, A.: *Eliminating Redundant Interleavings during Concurrent Program Verification.* Proceedings of Parallel Architectures and Languages Europe '89 Vol 2, Lecture Notes in Computer Science 366, Springer 1989 pp. 89–103.

[Valmari 89b] Valmari, A.: *Stubborn Sets for Reduced State Space Generation.* Proceedings of the Tenth International Conference on Application and Theory of Petri Nets, Bonn, FRG 1989 Vol. 2 pp. 1–22. A revised version to appear in Advances in Petri Nets 90, Lecture Notes in Computer Science, Springer.

[Wheeler & 90] Wheeler, G. R., Valmari, A. & Billington, J.: *Baby Toras Eats Philosophers but Thinks about Solitaire.* Proceedings of the Fifth Australian Software Engineering Conference, Sydney, NSW, Australia, 1990 pp. 283–288.