The copyright of this text belongs to Springer-Verlag. The text is re-published (possibly with small layout changes) in this web page according to the permission stated in http://www.springer.de/comp/lncs/copyrigh.html
The homepage of the LNCS series is http://www.springer.de/comp/lncs/index.html

Proceedings of CAV '93, 5th International Conference on Computer-Aided Verification, Elounda, Greece, Lecture Notes in Computer Science 697, Springer-Verlag 1993, pp. 397-408.

ON-THE-FLY VERIFICATION WITH STUBBORN SETS

Antti Valmari

Tampere University of Technology
Software Systems Laboratory
PO Box 553, SF-33101 Tampere, FINLAND
ava@cs.tut.fi

&

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

Abstract

A new on-the-fly verification method is presented. The method uses a generalization of Büchi automata called "tester processes" for representing and detecting illegal behaviour. To reduce the number of states that are constructed the method applies the stubborn set theory in a new way. The method can be used in connection with the "Supertrace" memory-saving technique. A simple algorithm is suggested for efficient detection of violations of an important subclass of liveness properties during the construction of the reduced state space.

1 INTRODUCTION

Many verification tools for concurrent systems are based on the construction of some variant of a *state space*. By "state space" we mean a structure consisting of the reachable states of the system and transitions that take the system from one state to another. Depending on the application, different aspects of the state space are emphasized, and it may also be called *labelled transition system* or *Kripke structure*. Various verification algorithms can be applied to the completed state space. Another, usually more efficient but less flexible approach is to perform verification *on-the-fly*, that is, simultaneously with the construction of the state space. With on-the-fly verification, construction of the state space can be stopped after the first violation against the specification has been found.

Obviously, when trying to verify an incorrect system, on-the-fly verification can be much more efficient than ordinary verification, because the number of states investigated before finding the first error can be much smaller than the total number of states. Of course, this does not help with correct systems. However, correct systems often obey some operational principles or invariants which significantly restrict the number of states they can reach. Incorrect systems violate those invariants. Therefore, the number of states of an incorrect system may be significantly higher than the number of states of the corresponding correct system. Consequently, on-the-fly verification often offers improved efficiency exactly where it is needed most. Furthermore, on-the-fly verification renders possible to avoid the investigation of branches of state spaces which cannot contain errors. For instance, when verifying the property "every execution starting with *a* continues with *b*" an on-the-fly verifier need not investigate further a sequence that started with *c*.

One more advantage of on-the-fly methods is that with them transition information need not always be stored. This also facilitates their use in connection with memory saving algorithms performing an incomplete state space search, most notably Holzmann's "Supertrace" algorithm [Holzmann 91]. Such algorithms are not exactly verification algorithms because they are not guaranteed to find all errors, but they are certainly an interesting practical alternative when the verification problem is too big for proper verification algorithms.

The use of on-the-fly verification for detecting deadlocks and improper single states or events (e.g. reception of an unexpected message) is straightforward. Violations of more complicated safety properties can be detected by encoding the property as a tester process which issues a single-state error condition when it has detected an improper sequence of actions. This, too, is standard technology. An algorithm which facilitates the use of on-the-fly verification for arbitrary linear temporal properties was described in [Courcoubetis & 90]. It uses so-called *Büchi automata* as tester processes. The basic idea of a Büchi automaton is that a sequence of actions is accepted (actually, deemed illegal in this application) if it may cause the automaton to go infinitely often through some state in a pre-specified set of states. Also multiple acceptance sets (i.e., each set should be visited infinitely often) can be treated with the techniques in [Courcoubetis & 90]. In the article, the Büchi automaton is synchronized with every transition of the system under analysis.

The *stubborn set* method is another technique for improving the efficiency of state space -based verification. It is based on the fact that in concurrent systems, the total effect of a set of actions is often independent of the order in which the actions are taken. The stubborn set method has been developed and explained in more than ten works since 1988, the most important of which are [Valmari 88, 89a, 89b, 90 and 92]. Related ideas have been presented by other researchers, including [Godefroid 90, & 91a, & 91b] and [Holzmann & 92].

The first application of stubborn sets was the detection of deadlocks and non-termination [Valmari 88, 89a]. This algorithm can easily be applied to on-the-fly verification. In [Valmari 89b] stubborn sets were used to detect illegal single events. The technique developed there is based on on-the-fly division of the state space into strong components. So it is amenable to on-the-fly detection of illegal events, but it cannot easily be used in connection with "Supertrace". Later stubborn sets were used, among other things, to construct reduced state spaces which are "as good as" the ordinary state spaces for the verification of stuttering-invariant linear temporal properties [Valmari 90], and CSP [Hoare 85] and CFFD [Valmari & 91] process algebraic semantic models [Valmari 92]. These techniques are not on-the-fly.

The article [Godefroid 90] presents an on-the-fly verification method for checking language containment, where both languages are represented by Petri nets generating them. The method is based on somewhat strict assumptions about the Petri net specifying the larger language. The paper [Godefroid & 91a] tries to combine Godefroid's version of stubborn set approach to Büchi automata to obtain an on-the-fly stubborn set -like method for general linear temporal properties. However, the combination proved problematic; the algorithm in the paper does not work as intended and has been modified later. Also, not all subtasks within the algorithm have been given on-the-fly solutions. In [Godefroid & 91b] general safety properties are verified with a technique suitable for on-the-fly use and "Supertrace". The error detection problem is transformed to the deadlock detection problem by using tester processes, and by adding extra transitions to every loop of certain processes of the system. Another, simpler and perhaps also more efficient technique suitable for safety properties was presented in [Holzmann & 92]. It is related to and apparently less powerful than the technique in [Valmari 89b], because the latter performs a more careful and thus less wasteful analysis for ensuring that every process is investigated "far enough". The [Holzmann & 92] technique has, however, the advantage of being well amenable to "Supertrace".

The goal of this article is to combine stubborn sets and on-the-fly verification better than before. Firstly, we use a new technique for ensuring that every process is investigated "far enough": we let the property being verified guide the investigation. We believe that this new technique makes our algorithm more powerful than those in [Holzmann & 92] and [Valmari 89b], while still applicable with "Supertrace". Secondly, we present what we believe to be a working combination of stubborn set -like techniques and Büchi automata. The combination can be used in connection with the [Courcoubetis & 90] algorithm and "Supertrace". The key problem in such a combination is that synchronizing the Büchi automaton to every action of the system makes all transitions dependent on each other and, consequently, prevents any reduction obtainable from the use of independency. On the other hand, incomplete synchronization creates fairness problems between the system and the Büchi automaton. We solve these problems by using a special kind of acceptance states for infinite sequences in which the system runs forever without synchronization, and by forbidding the Büchi automaton from running around forever on its own. Thirdly, we present a simple algorithm for checking certain liveness properties. The algorithm is less powerful than the [Courcoubetis & 90] algorithm, but it is (we believe) simpler and faster, and it is powerful enough to handle divergence detection in the sense of CSP and CFFD theories. For simplicity, in this article we concentrate on the verification of properties which talk about action sequences, although our techniques can be applied to stuttering-invariant properties of state sequences as well.

2 THEORETICAL BACKGROUND

In this work we assume that a concurrent system can be thought of as a collection of parallel *processes*. Each process has an *alphabet* listing the set of *actions* it is interested in, and processes communicate by executing synchronously common actions. The behaviour of a process may be represented as a structure known as the *state space*, which consists of all states of the process together with all transitions the process can make between those states. For simplicity, we do not bother to enforce a distinction between a process and its state space, but define a graph-like structure which we call "process", "state space", or "labelled transition system" depending on the context.

Definition 2.1 A *process* is the quadruple (S, Σ, Δ, si), where

- S is a finite set of *states*,
- Σ is a finite *alphabet*,
- $\Delta \subseteq S \times \Sigma \times S$ is the set of *transitions*, and
- $si \in S$ is the *initial state*.

[]

Notation for talking about finite and infinite action sequences a process can execute, and the set of states it can reach, is introduced in the following definition.

Definition 2.2 Let $P = (S, \Sigma, \Delta, si)$ be a process.

- $s \xrightarrow{a} s'$ iff $(s, a, s') \in \Delta$.
- $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ iff $s_{i-1} \xrightarrow{a_i} s_i$ for $1 \le i \le n$.
- $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots$ iff $s_{i-1} \xrightarrow{a_i} s_i$ for $i \ge 1$.
- $s \xrightarrow{a_1 a_2 \dots a_n} s'$ iff there are $s_0, s_1, \dots, s_n$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$, $s_0 = s$, and $s_n = s'$.
- $s \xrightarrow{a_1 a_2 \dots a_n}$ iff there is s' such that $s \xrightarrow{a_1 a_2 \dots a_n} s'$.
- $s \xrightarrow{a_1 a_2 \dots}$ iff there are $s_0, s_1, \dots$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots$ and $s_0 = s$.
- $reach(P) = \{ s \in S \mid \exists n \ge 0 \colon \exists a_1, a_2, \dots, a_n \in \Sigma \colon si \xrightarrow{a_1 a_2 \dots a_n} s \}$.
- $norm(P) = (S', \Sigma, \Delta', si)$, where $S' = reach(P)$ and $\Delta' = \{ (s, a, s') \in \Delta \mid s \in S' \}$.

[]

A system consisting of two or more processes connected in parallel may execute an action if and only if all processes interested in that action may execute it.

Definition 2.3 Let $P_1 = (S_1, \Sigma_1, \Delta_1, si_1)$, $P_2 = (S_2, \Sigma_2, \Delta_2, si_2)$, …, $P_h = (S_h, \Sigma_h, \Delta_h, si_h)$ be processes. The *parallel composition* of $P_1, P_2, \dots, P_h$ is denoted by $P_1 \parallel P_2 \parallel \dots \parallel P_h$, and it is the process (S, Σ, Δ, si) such that

- $S = S_1 \times S_2 \times \dots \times S_h$

- $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_h$
- $(s, a, s') \in \Delta$ where $s = (s_1, s_2, \ldots, s_h) \in S$, $s' = (s'_1, s'_2, \ldots, s'_h) \in S$ and $a \in \Sigma$ if and only if

  $(s_i, a, s'_i) \in \Delta_i$ for all i such that $a \in \Sigma_i$, and
  $s_i = s'_i$ for other values of i
- $si = (si_1, si_2, \ldots, si_h)$

[]

We assume that not all actions of a process are directly interesting from the point of view of the properties that we wish to verify. Therefore, we assume that a set Σ_{vis} of *visible* actions has been defined. The properties of a process that we are interested in are stated in terms of the sequences of visible actions corresponding to execution sequences of the process, together with some indirect information about invisible actions. We denote the empty string by ε, and A^* and A^∞ denote the sets of finite and infinite strings of elements of A, respectively.

Definition 2.4 Let $P = (S, \Sigma, \Delta, si)$ be a process, and $\Sigma_{vis} \subseteq \Sigma$ the set of visible actions. Let $\sigma \in \Sigma_{vis}^*$, $\xi \in \Sigma_{vis}^* \cup \Sigma_{vis}^\infty$, and $A \subseteq \Sigma_{vis}$.

- If $\rho \in \Sigma^* \cup \Sigma^\infty$, then $vis(\rho)$ is ρ with all elements not in Σ_{vis} removed.
- $s =\sigma\Rightarrow s'$ iff $\exists\, \rho \in \Sigma^*$: $s \text{—}\rho\text{→}\, s' \wedge vis(\rho) = \sigma$.
- $s =\xi\Rightarrow$ iff $\exists\, \rho \in \Sigma^* \cup \Sigma^\infty$: $s \text{—}\rho\text{→} \wedge vis(\rho) = \xi$.
- If $s \in S$, then $next(P, s) = \{\, a \in \Sigma \mid s \text{—}a\text{→} \,\}$. If P is obvious from the context, we may write $next(s)$ instead of $next(P, s)$.
- $tr(P) = \{\, \sigma \in \Sigma_{vis}^* \mid si =\sigma\Rightarrow \,\}$ is the set of *traces* of P.
- $inftr(P) = \{\, \xi \in \Sigma_{vis}^\infty \mid si =\xi\Rightarrow \,\}$ is the set of *infinite traces* of P.
- $divtr(P) = \{\, \sigma \in \Sigma_{vis}^* \mid \exists\, \rho \in \Sigma^\infty$: $s \text{—}\rho\text{→} \wedge vis(\rho) = \sigma \,\}$ is the set of *divergence traces* of P. Equivalently,
  $divtr(P) = \{\, \sigma \in \Sigma_{vis}^* \mid \exists\, s$: $si =\sigma\Rightarrow s \wedge \exists\, \rho \in (\Sigma - \Sigma_{vis})^\infty$: $s \text{—}\rho\text{→} \,\}$.
- $sfail(P) = \{\, (\sigma, A) \mid A \subseteq \Sigma_{vis} \wedge \exists\, s$: $si =\sigma\Rightarrow s \wedge next(s) \subseteq \Sigma_{vis} - A \,\}$ is the set of *stable failures* of P.

[]

Except two remarks, our notions of traces etc. are standard notions in process algebraic theories. First, process algebraic theories typically use one special symbol "τ" to denote all actions that are not visible, and there is a special *hiding* operator which transforms visible actions to τ. In our context, however, it is technically simpler (e.g. Definitions 2.3 and 2.5) to allow distinct names for invisible actions, and to specify explicitly which actions are invisible. This does not imply loss of generality, because all τ:s can be made unique by subscribing them with the name of the process executing the τ (details of this are explained in [Valmari 92]). Second, from the two commonly used notions of failures we chose the *stable* one. That is, it is not possible to execute an invisible action in the state s in the definition of *sfail*.

Traces of P correspond to finite executions of P. Infinite traces and divergence traces correspond to infinite executions with infinite and finite numbers of visible actions, respectively. Stable failures give information about the ability of P to deadlock in different environments. In particular, an isolated P can deadlock after the trace σ if and only if $(\sigma, \Sigma_{vis}) \in sfail(P)$.

In the remainder of this section we repeat those facts of the stubborn set theory which are important for understanding the rest of this article. A detailed account of the stubborn set method for synchronous systems is found in [Valmari 92]. First the definition of stubborn sets:

Definition 2.5 Consider the system $P_1 \parallel P_2 \parallel \ldots \parallel P_h$ with the notation from Definition 2.3. Let $s = (s_1, s_2, \ldots, s_h) \in S$ and $\ddot{A} \subseteq \Sigma$.

- $\ddot{A}$ is *semistubborn* at s, iff $\forall\, a \in \ddot{A}$: (1) $\wedge$ (2), where
  (1) $\neg(\, s \text{ —}a\text{→}\,) \;\Rightarrow\; \exists\, i \in \{1, 2, \ldots, h\}$: $a \in \Sigma_i \wedge a \notin next(P_i, s_i) \wedge next(P_i, s_i) \subseteq \ddot{A}$
  (2) $s \text{ —}a\text{→} \;\Rightarrow\; \forall\, i \in \{1, 2, \ldots, h\}$: $a \notin \Sigma_i \vee next(P_i, s_i) \subseteq \ddot{A}$
- $\ddot{A}$ is *stubborn* at s, iff it is semistubborn at s, and $\exists\, a \in \ddot{A}$: $s \text{ —}a\text{→}$.

[]

That is, a semistubborn set is a set of actions designed so that (1) to enable a disabled action in the set it is necessary to execute at least one action in the set, and (2) enabled actions in the set are independent of (i.e. participated by separate processes from) enabled actions outside the set. Also, a (semi)stubborn set remains (semi)stubborn when an outside action occurs. From these facts it is possible to show the following theorem, which says that actions belonging to a currently semistubborn set can be swapped with outside actions in certain ways.

Theorem 2.6 Consider the system $P_1 \parallel P_2 \parallel \ldots \parallel P_h = (S, \Sigma, \Delta, si)$. Let $s_0 \in S$, $\ddot{A} \subseteq \Sigma$ be a semistubborn set of actions at s_0, and let $a \in \ddot{A}$ and $a_1, a_2, \ldots, a_n, \ldots \notin \ddot{A}$.

- If $s_0 \text{ —}a_1a_2\ldots a_n\text{→ } s_n$ and $s_n \text{ —}a\text{→ } s'_n$,
  then there is s'_0 such that $s'_0 \text{ —}a_1a_2\ldots a_n\text{→ } s'_n$ and $s_0 \text{ —}a\text{→ } s'_0$.
- If $s_0 \text{ —}a_1a_2\ldots a_n\text{→ } s_n$ and $s_0 \text{ —}a\text{→ } s'_0$,
  then there is s'_n such that $s'_0 \text{ —}a_1a_2\ldots a_n\text{→ } s'_n$ and $s_n \text{ —}a\text{→ } s'_n$.
- If $s_0 \text{ —}a_1a_2\ldots\text{→}$ and $s_0 \text{ —}a\text{→ } s'_0$, then $s'_0 \text{ —}a_1a_2\ldots\text{→}$.

[]

Therefore, if $\ddot{A}$ is stubborn at the state we are currently at, then all states reachable through an action sequence containing at least one action from $\ddot{A}$ are reachable even if we currently execute only actions from $\ddot{A}$ and postpone the execution of other actions. This idea is formalized in the following construction of a *reduced state space*.

Definition 2.7 Let $P = P_1 \parallel P_2 \parallel \ldots \parallel P_h = (S, \Sigma, \Delta, si)$ be a parallel composition of processes. A *stubborn set generator* of $P_1 \parallel P_2 \parallel \ldots \parallel P_h$ is a function $\ddot{A}$: $S_1 \times S_2 \times \ldots \times S_h \to 2^{\Sigma}$ such that if s is not a deadlock state of P, then $\ddot{A}(s)$ is stubborn at s. We write $s \text{ \textasciitilde}a\text{\textasciitilde> } s'$, if $(s, a, s') \in \Delta$ and $a \in \ddot{A}(s)$. The *reduced state space* of P determined by $\ddot{A}$ is the 4-tuple $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$, where

- $S_{\ddot{A}}$ is the smallest set such that $si \in S_{\ddot{A}}$, and, if $s \in S_{\ddot{A}}$ and s ~a~> s', then $s' \in S_{\ddot{A}}$.
- $\Delta_{\ddot{A}} = \{\ (s, a, s') \in S_{\ddot{A}} \times \Sigma \times S_{\ddot{A}} \mid s$ ~a~> $s'\ \}$

[]

That is, the reduced state space of a system is otherwise the same as the ordinary state space of the system, but at every state s, only the actions in $\ddot{A}(s)$ are used to construct output transitions and successor states of s. It is fairly obvious from the construction that any reduced state space of a system is contained by the ordinary state space of the system.

Theorem 2.8 If $norm(P) = (S', \Sigma, \Delta', si)$, and $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$ is the reduced state space of P determined by a stubborn set generator $\ddot{A}$ of P, then $S_{\ddot{A}} \subseteq S'$ and $\Delta_{\ddot{A}} \subseteq \Delta'$. []

What can we say about a system, given its reduced state space? Actually, not much. The reduced state space as defined above is guaranteed to preserve only termination-oriented properties of the system.

Theorem 2.9 Let $P = (S, \Sigma, \Delta, si)$, and let $P_{\ddot{A}} = (S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$ be a reduced state space of P.

- $s \in reach(P)$ and $next(P, s) = \emptyset$ if and only if $s \in S_{\ddot{A}}$ and $next(P_{\ddot{A}}, s) = \emptyset$
  (i.e. the ordinary and reduced state space contain exactly the same deadlocks).

- P may execute an infinite action sequence if and only if $P_{\ddot{A}}$ contains an infinite action sequence.

[]

Properties other than those given in Theorem 2.9 cannot be decided from the reduced state space because of the so-called "ignoring problem": after detecting that some component processes may run in a loop without synchronizing with the rest of the system, the construction in Definition 2.7 may terminate without investigating the remaining processes. To ensure that all processes are investigated "far enough" it is necessary to further constrain the reduced state space construction. Some ways of doing this were mentioned in the introduction. One more approach, tuned directly to the needs of on-the-fly verification, will be given in Theorem 4.1.

3 TESTER PROCESSES

We intend to check against four kinds of unwanted properties of a process:

- illegal traces,
- illegal infinite traces,
- illegal divergence traces, and
- illegal stable failures.

Because any prefix of a trace is a trace, we will treat a continuation of an illegal trace as illegal. Similarly, if (σ, A) is an illegal stable failure and $A \subseteq B$, then (σ, B) is an illegal stable failure.

We will represent the illegal traces etc. with the aid of a *tester process*. A tester process resembles an ordinary process, but its states have special meanings that are described soon. The verification will be performed by connecting the tester process in parallel with the system, constructing a subset of the reachable states of the combination, and analysing the states and their relationships in a certain way explained soon. The formal definition of tester process is as follows:

Definition 3.1 Let Σ_{vis} be the set of visible actions. The tuple $T = (S_T, \Sigma_T, \Delta_T, si_T, S_O, S_R, S_D, S_L, S_\infty)$ is a *tester*, if the following hold:

- $P_T = (S_T, \Sigma_T, \Delta_T, si_T)$ is a process.
- $\Sigma_T = \Sigma_{vis} \cup \{\tau\}$, where τ is a so far unused symbol.
- $S_R \cup S_D \cup S_L \cup S_\infty \subseteq S_T$, and $S_O = S_T - (S_R \cup S_D \cup S_L \cup S_\infty)$.
- Δ_T contains no τ-loops, i.e. there are no $n \geq 1$ and $s_0, s_1, \ldots, s_n \in S_T$ such that $(s_0, \tau, s_1), (s_1, \tau, s_2), \ldots, (s_{n-1}, \tau, s_n) \in \Delta_T$ and $s_n = s_0$.
- If $(s, \tau, s') \in \Delta_T$, then $s \notin S_D$, i.e. states in S_D have no τ-labelled output transitions.

The elements of S_O to S_∞ are called *ordinary states*, *reject states*, *deadlock monitor states*, *livelock monitor states*, and *infinite trace monitor states*, respectively. []

Reject states, deadlock monitor states, livelock monitor states and infinite trace monitor states are used to specify and detect illegal traces, illegal stable failures, illegal divergence traces and illegal infinite traces, respectively.

Definition 3.2 Let T be a tester. We use the notation from Definition 3.1. Let $\sigma \in \Sigma_{vis}^*$, $\xi \in \Sigma_{vis}^\infty$ and $A \subseteq \Sigma_{vis}$.

- σ is an *illegal trace*, iff it has a prefix σ' such that $\exists\, s \in S_T$: $si_T = \sigma' \Rightarrow s \wedge s \in S_R$.
- σ is an *illegal divergence trace*, iff $\exists\, s \in S_T$: $si_T = \sigma \Rightarrow s \wedge s \in S_L$.
- (σ, A) is an *illegal stable failure*, iff $\exists\, s \in S_T$: $si_T = \sigma \Rightarrow s \wedge s \in S_D \wedge next(s) \subseteq A$.

- ξ is an *illegal infinite trace*, iff there are $s_0, s_1, s_2, \ldots \in S_T$ and $a_1, a_2, \ldots \in \Sigma_T$ such that $s_0 = si$, $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots$, $vis(a_1 a_2 \ldots) = \xi$, and $s_i \in S_\infty$ for infinitely many values of i.

[]

The following result is fairly obvious from Definitions 3.1 and 3.2, and the fact that we required that the number of states of a process is finite. It gives necessary and sufficient conditions for detecting violations by P_S against the specification represented by T from the state space of $P_T \parallel P_S$.

Theorem 3.3 Let $P_S = (S_S, \Sigma_S, \Delta_S, si_S)$ be a process, and T a tester such that $\tau \notin \Sigma_S$ and $\Sigma_{vis} \subseteq \Sigma_S$. We use the notation from Definition 3.1. Let **C1** to **C4** be the following conditions on $P = P_T \parallel P_S$:

C1 There is $(s_T, s_S) \in reach(P)$ such that $s_T \in S_R$.

C2 There is $s = (s_T, s_S) \in reach(P)$ such that $s_T \in S_D$ and $next(P, s) = \emptyset$.
(That is, a deadlock state (s_T, s_S) is reachable such that $s_T \in S_D$.)

C3 There are $n \geq 1$, $s_T \in S_L$, $s_{S0}, s_{S1}, \ldots, s_{Sn} \in S_S$, and $a_1, a_2, \ldots, a_n \in \Sigma_S - \Sigma_{vis}$ such that $(s_T, s_{S0}) \in reach(P)$, $s_{Sn} = s_{S0}$, and $(s_T, s_{Si-1}) \xrightarrow{a_i} (s_T, s_{Si})$ for $1 \leq i \leq n$.
(That is, a loop is reachable such that all transitions in it correspond to $\Sigma_S - \Sigma_{vis}$, and at least one (equivalently, every) state (s_T, s_S) in it has the property that $s_T \in S_L$.)

C4 There are $n \geq 1$, $a_1, a_2, \ldots, a_n \in \Sigma_S \cup \{\tau\}$ and $(s_{T0}, s_{S0}), (s_{T1}, s_{S1}), \ldots, (s_{Tn}, s_{Sn}) \in reach(P)$ such that $s_{Tn} = s_{T0}$, $s_{Sn} = s_{S0}$, $(s_{Ti-1}, s_{Si-1}) \xrightarrow{a_i} (s_{Ti}, s_{Si})$ for $1 \leq i \leq n$, $s_{T0} \in S_\infty$, and $a_1 \in \Sigma_{vis} \cup \{\tau\}$.
(That is, a loop is reachable such that at least one transition in the loop corresponds to $\Sigma_{vis} \cup \{\tau\}$, and the start state (s_T, s_S) of that transition has the property that $s_T \in S_\infty$. Note that due to the "no τ-loops" assumption in Definition 3.1, every loop of P containing a τ-transition contains also an a-transition such that $a \in \Sigma_{vis}$.)

We have

- $tr(P_S)$ contains an illegal trace if and only if **C1** holds.
- $sfail(P_S)$ contains an illegal stable failure if and only if **C2** holds.
- $divtr(P_S)$ contains an illegal divergence trace if and only if **C3** holds.
- $inftr(P_S)$ contains an illegal infinite trace if and only if **C4** holds.

[]

Some words about the relationship between testers and Büchi automata are perhaps appropriate here. A Büchi automaton can be straightforwardly simulated by a tester, by letting the states corresponding to the accept states of the Büchi automaton to be simultaneously both livelock monitor states and infinite trace monitor states. The fact that a tester process P_T is not synchronised to every action of the system P_S has the consequence that infinite executions of P_S correspond to two kinds of infinite executions of $P = P_T \parallel P_S$: those which are participated infinitely often by P_T, and those which are not. This distinction is reflected in testers by having two kinds of infinite accept states, namely S_∞ and S_L. To simulate a Büchi automaton this distinction need not be enforced.

In our context the above mentioned distinction is useful, however, because the two kinds of loops correspond to two different kinds of errors. It can be shown that if legal traces etc. are specified by giving a finite state process such that exactly its traces etc. are legal, then another finite state process cannot have illegal infinite traces without having also illegal finite traces. That is, the only way to introduce illegal infinite traces without introducing illegal finite traces is to add some fairness assumptions to the specification process. (Therefore, we may divide liveness properties to "ordinary" and "fair" according to whether their violations appear as **C3** or **C4** errors.) So, in the absence of fairness assumptions, the ability to test illegal infinite trace errors is not needed. This is an advantage because, as we show next, illegal divergence traces can be tested with a simpler algorithm than illegal infinite traces. By the way, a finite state process can be converted to a tester algorithmically, by applying finite automaton determinization construction, and the canonical tester theory from [Brinksma 88] in a certain way.

Algorithm 3.4 below can be used for efficient on-the-fly detection of error conditions **C1**, **C2** and **C3**. **C1** and **C2** can obviously be checked from each reachable state separately. To check **C3** the algorithm performs depth-first searches using only the transitions $(s_T, s_S) \xrightarrow{a} (s'_T, s'_S)$ such that $a \in \Sigma_S - \Sigma_{vis}$ and $s_T \in S_L$. During such a depth-first search also other transitions are investigated to ensure that the whole state space is eventually covered, but, in order to not confuse the depth-first search order, the resulting states are neither entered nor marked found at this stage. Instead, they are stored into the set *mainset*, from which they are one by one taken to investigation when the previous depth-first searches have terminated. The *stack* variable contains the search stack of the depth-first searches. The "Supertrace" hash table technique can be used to keep track of which states have been marked "found", if a memory saving incomplete verification algorithm is desired. Algorithm 3.4 is closely related to Algorithm 3 in [Valmari 90] and Algorithm 4.19 in [Valmari 92].

Algorithm 3.4
(* See Definition 3.1, Theorem 3.3, and text for explanation of notation. *)

```
$stack := \emptyset$; $mainset := \{si\}$; $checkC1C2(si)$;    (* $si = (si_T, si_S)$ *)
while $mainset \neq \emptyset$ do
    let $s \in mainset$; $mainset := mainset - \{s\}$;
    if not found($s$) then push $s$ to $stack$; mark $s$ found endif;
    while $stack \neq \emptyset$ do
        let $s = (s_T, s_S)$ = top($stack$);
        if there is an uninvestigated $(s, a, s')$ such that $s \xrightarrow{a} s'$ then
            mark $(s, a, s')$ investigated;
            if not found($s'$) then
                $checkC1C2(s')$;
                if $a \in \Sigma_S - \Sigma_{vis}$ and $s_T \in S_L$ then
                    push $s'$ to $stack$; mark $s'$ found
                else
                    $mainset := mainset \cup \{s'\}$
                endif
            elsif $a \in \Sigma_S - \Sigma_{vis}$ and $s_T \in S_L$ and $s' \in stack$ then
                report "C3 error"; halt
            endif
        else
            pop s from stack
        endif
    endwhile
endwhile
procedure $checkC1C2( (s_T, s_S)$ : state ) is
    if $s_T \in S_R$ then report "C1 error"; halt
    elsif $s_T \in S_D$ and $next(P, (s_T, s_S)) = \emptyset$ then report "C2 error"; halt
    endif
endproced
```

[]

To check the error condition **C4** one has to find or demonstrate the absence of a reachable loop containing a transition in a certain set Δ_{C4} of transitions. In [Courcoubetis &

90] an algorithm is given for on-the-fly detection of a reachable loop going through a state in a certain set S_{C4} of states. This algorithm can be adapted to our problem simply by pretending that each transition in Δ_{C4} has an imaginary middle state, and S_{C4} consists of these middle states. Of course, also the [Courcoubetis & 90] algorithm can be decorated with calls to *checkC1C2*, to detect violations of **C1**, **C2** and **C4** in the same run.

4 USE OF STUBBORN SETS WITH TESTER PROCESSES

The following theorem shows that if the stubborn sets used during reduced state space generation are selected in a suitable way, then at least one of the conditions **C1** to **C4** from Theorem 3.3 holds in the reduced state space if and only if at least one of them holds in the ordinary state space. Consequently, as stated in the corollary after the theorem, the reduced state space can be used to verify that the system satisfies the properties represented by the tester, or to find *one* violation against the properties. The theorem and corollary do not guarantee that *all* violations, or even an example of *every kind* of violations are found, however, but this is not usually needed anyway.

Theorem 4.1 Let $P_S = P_1 \parallel P_2 \parallel \ldots \parallel P_h = (S_S, \Sigma_S, \Delta_S, si_S)$ be a parallel composition of processes. Let T be a tester such that $\tau \notin \Sigma_S$ and $\Sigma_{vis} \subseteq \Sigma_S$, and the notation from Definition 3.1 is used. Let $\ddot{A}$ be a stubborn set generator of $P = P_T \parallel P_1 \parallel P_2 \parallel \ldots \parallel P_h$, and let $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$ be the reduced state space of P determined by $\ddot{A}$. Let **C***i*, $1 \leq i \leq 4$, be as in Theorem 3.3. If for every $s = (s_T, s_1, s_2, \ldots, s_h) \in S_{\ddot{A}}$

(1) If $s_T \notin S_L$, then $\Sigma_T \subseteq \ddot{A}(s)$.

(2) If $s_T \in S_L$ and there is $a \in \Sigma_S - \Sigma_{vis}$ such that $s \xrightarrow{a}$,
then there is $a \in \ddot{A}(s) \cap (\Sigma_S - \Sigma_{vis})$ such that $s \xrightarrow{a}$.

then

- If **C***i* holds for $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$, then **C***i* holds for P.
- If **C***i* holds for P, then **C***i* or **C3** holds for $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$.

Proof The first claim is easy: if any of **C***i* holds for $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$, then by Theorem 2.8 it holds also for P. In the case of **C2**, the first part of Theorem 2.9 is also used.

The details of the proof of the second claim are omitted due to the length of the proof. The basic argument is the following: given an execution $(s_{T0}, s_{S0}) \xrightarrow{a_1} (s_{T1}, s_{S1}) \xrightarrow{a_2} \ldots$ where at least one a_k belongs to Σ_{vis}, two things can happen. If $s_{T0} \notin S_L$, then assumption (1) of the theorem forces the stubborn set method to execute one of $a_1, \ldots, a_k$. This guarantees that P_T eventually proceeds. If $s_{T0} \in S_L$, then either only a finite number of $a \in \Sigma_S - \Sigma_{vis}$ are taken before P_T proceeds, in which case P_T eventually proceeds; or an infinite number of them are taken, in which case **C3** holds. So, as long as **C3** does not hold, P_T is guaranteed to proceed in the reduced state space as far as necessary to detect any of **C***i*. In the cases of **C1** and **C4** this is sufficient. Regarding **C3**, after P_T has reached an s_T such that $s_T \in S_L$, it is necessary for P_S to proceed alone until a loop is found. Assumption (2) guarantees that this will happen if it is possible. The case of **C2** comes directly from Theorem 2.9 (it does not need the **C3**-option). []

Corollary 4.2 Let P_S, T, and $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$ be as in Theorem 4.1. The system P_S satisfies the specification represented by T if and only if none of **C1** to **C4** holds in $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$. []

To check efficiently on-the-fly whether any of **C1** to **C4** holds in $(S_{\ddot{A}}, \Sigma, \Delta_{\ddot{A}}, si)$, one has to combine the use of stubborn sets to Algorithm 3.4 and to the [Courcoubetis & 90] algorithm. This is done by restricting the investigation of output transitions of each state s to the stubborn set $\ddot{A}(s)$. In the case of Algorithm 3.4 this is done by replacing the line

```
    if there is an uninvestigated $(s, a, s')$ such that $s \xrightarrow{a} s'$ then
```

by the line

if there is an uninvestigated (s, a, s') such that $s \xrightarrow{a} s'$ and $a \in \ddot{A}(s)$ **then**.

To implement the stubborn set on-the-fly verification method suggested by Corollary 4.2, Algorithm 3.4 and the [Courcoubetis & 90] algorithm, an efficient algorithm is needed for one more task: the computation of $\ddot{A}(s)$. The construction of stubborn sets satisfying certain properties has been discussed in detail in [Valmari 90] and [Valmari 92]. We now repeat the basic ideas and then adapt them to the needs of Theorem 4.1.

With little loss of generality, the stubborn set construction problem can be rewritten as the problem of finding certain kinds of subgraphs of a graph $G(s) = (V, E_s)$ derived from the system $P_T \parallel P_1 \parallel P_2 \parallel \ldots \parallel P_h$ and its current state $s = (s_T, s_1, s_2, \ldots, s_h)$. The set of vertices of $G(s)$ is $V = \Sigma \cup \{T, 1, 2, \ldots, h\}$. The edges E_s of $G(s)$ correspond directly to (1) and (2) of Definition 2.5 as follows. Let $a \in \Sigma$. First, if $s \xrightarrow{a}$, then there is an edge from a to every $i \in \{T, 1, 2, \ldots, h\}$ such that $a \in \Sigma_i$. Second, if $\neg(s \xrightarrow{a})$, then there is an edge from a to an arbitrarily chosen $i \in \{T, 1, 2, \ldots, h\}$ such that $a \in \Sigma_i$ and $a \notin next(P_i, s_i)$ (the arbitrariness here causes the "little loss of generality" mentioned above). Finally, from each $i \in \{T, 1, 2, \ldots, h\}$ there is an edge to every $a \in next(P_i, s_i)$. The graph $G(s)$ has $O(h + |\Sigma| + 1)$ vertices and $O((h + 1) \cdot |\Sigma|)$ edges.

For any vertex a of $G(s)$, let *closure*(a) denote the set of vertices of $G(s)$ reachable from a in $G(s)$. Also, if $A \subseteq V$, then *closure*(A) is the union of *closure*(a) for $a \in A$. Due to the construction of $G(s)$, if $a \in \Sigma$ and $s \xrightarrow{a}$, then *closure*$(a) \cap \Sigma$ is stubborn. Of course, we want to keep the number of enabled actions in a stubborn set small to obtain a small reduced state space. Unfortunately, the set *closure*$(a) \cap \Sigma$ for an arbitrary enabled a is not necessarily very good in this respect. Much better stubborn sets can be found by analysing the strong components of $G(s)$. Strong components can be found in depth-first order by an algorithm described in [Aho & 74] Section 5.5. The first strong component containing an enabled action a' found by the algorithm has the property that no other strong component reachable from it contains enabled actions. Therefore, *closure*$(a') \cap \Sigma$ is a locally optimal stubborn set among those represented by $G(s)$. Similarly, a locally $G(s)$-optimal stubborn set containing an enabled action $a \in \Sigma_S - \Sigma_{vis}$ can be found by searching for a strong component until one containing an enabled action from $\Sigma_S - \Sigma_{vis}$ is found. These considerations lead to the following algorithm implementing $\ddot{A}(s)$.

Algorithm 4.3

```
function $\ddot{A}(s = (s_T, s_S) : \text{state})$ : set of actions is
    if $s_T \notin S_L$ then
        $\ddot{A} := closure(\Sigma_T) \cap \Sigma$
    elsif $next(P, s) \cap (\Sigma_S - \Sigma_{vis}) \neq \emptyset$ then
        run the strong component search algorithm until a strong component is found
        such that it contains an enabled action $a \in \Sigma_S - \Sigma_{vis}$;
        $\ddot{A} := closure(a) \cap \Sigma$
    else $\ddot{A} := \emptyset$
    endif;
    if $\ddot{A} \cap next(P, s) = \emptyset$ and $next(P, s) \neq \emptyset$ then
        run the strong component search algorithm until a strong component is found
        such that it contains an enabled action $a$;
        $\ddot{A} := \ddot{A} \cup (closure(a) \cap \Sigma)$
    endif;
    return $\ddot{A}$
endfunction
```

[]

The above algorithm first tests for the conditions in assumptions (1) and (2) of Theorem 4.1. If either of the conditions holds, the algorithm computes a semistubborn set guided by the assumption. This set is not necessarily stubborn, however, because it is not guaranteed to contain an enabled action. Furthermore, it is not even computed, if $s_T \in S_L$ and $next(P, s) \cap (\Sigma_S - \Sigma_{vis}) = \emptyset$. Therefore, the algorithm finally computes a truly stubborn set if necessary ($\ddot{A} \cap next(P, s) = \emptyset$), and possible ($next(P, s) \neq \emptyset$). Because the strong component search algorithm consumes $O(V + E_S)$ time and the closures can be computed by graph traversal, Algorithm 4.3 can be implemented to run in $O((h+1) \cdot |\Sigma|)$ time.

5 CONCLUSIONS

We presented an on-the-fly stubborn set verification method for four kinds of properties of systems. The properties are represented as a tester process, with four kinds of different states for error detection. The tester process is synchronized with the system under analysis, and errors in the behaviour of the system are detected by searching for two kinds of states and two kinds of loops in the state space of the combination. We showed how stubborn sets can be used to construct a reduced state space of the combination such that it can replace the ordinary state space of the combination for error detection. We also gave an algorithm for on-the-fly detection of three kinds of errors, and showed how an algorithm in [Courcoubetis & 90] can be used to detect errors of the fourth class on-the-fly. These algorithms, including their stubborn set versions, can be used in connection with Holzmann's "Supertrace".

Our division of errors to four classes was motivated partly by the fact that each class has a natural meaning in process algebraic theories, and partly by the existence of separate verification algorithms for the two classes requiring loop detection. Although most of what can be expressed with our testers can be represented by Büchi automata (Büchi automata probably cannot fully distinguish between divergence and deadlock, which testers can do — see [Kaivola & 92]) and thus could be verified with only one type of error condition, we believe that our distinction is useful in that it allows certain properties to be verified more efficiently. Our algorithm for detecting divergence errors is expected to be faster than the [Courcoubetis & 90] algorithm on the same problem, because the former finds an illegal loop immediately when all transitions in it have been found, whereas the latter investigates fully the area below the loop in the search order before attempting loop detection. Furthermore, our algorithm searches the state space only once, while the [Courcoubetis & 90] algorithm does it twice. Of course, these arguments are only informal; we do not have any analytical or experimental results about the relative speeds of the two algorithms.

We solved the so-called "ignoring problem" of stubborn sets by requiring that the stubborn set contains all actions of the tester except when the tester is expecting an illegal livelock (assumption (1) of Theorem 4.1). This makes our approach very goal-oriented: the stubborn set selection favours actions which make progress towards an error condition over those which do not. The price we paid for this is that we have to repeatedly compute $\ddot{A}_T = closure(\Sigma_T) \cap \Sigma$ and include it to the stubborn set. However, its computation is relatively fast. Note that the inclusion of $\ddot{A}_T$ to the stubborn set does not mean that unduly many enabled actions are included. Instead, the enabled actions in $\ddot{A}_T$ are exactly those whose execution may eventually lead to progress of the tester, as far as the stubborn set method can know. The set $\ddot{A}_T$ may contain many disabled actions, but this is not a problem, because disabled actions do not introduce states into the reduced state space. However, their presence guides the method to select the stubborn set leading most directly to error conditions.

How good is our method? At the time of finishing this article no experimental or analytical results were available. But the not-on-the-fly versions of the stubborn set method have proven very powerful (e.g. reduction from $9n2^{n-2}$ to $5n$ states in a n-customer token ring problem posed by Graf and Steffen [Valmari 92]), and there is no reason why the addition of the on-the-fly idea to them should not be an improvement.

Acknowledgements

Part of this work was funded 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 1974, 470 p.

[Brinksma 88] Brinksma, E.: *A Theory for the Derivation of Tests*. Protocol Specification, Testing and Verification VIII (Proceedings of International IFIP WG 6.1 Symposium, 1988), North-Holland 1988, pp. 63–74.

[Courcoubetis & 90] Courcoubetis, C.; Vardi, M.; Wolper, P. & Yannakakis, M.: *Memory Efficient Algorithms for the Verification of Temporal Properties*. Formal Methods in System Design, 1: 275–288 (1992). (An earlier version appeared in Workshop on Computer-Aided Verification, New Brunswick, NJ, USA 1990.)

[Godefroid 90] Godefroid, P.: *Using Partial Orders to Improve Automatic Verification Methods*. AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science Volume 3, 1991, pp. 321–340. (An earlier version appeared in Workshop on Computer-Aided Verification, New Brunswick, NJ, USA 1990.)

[Godefroid & 91a] Godefroid, P. & Wolper, P.: *A Partial Approach to Model Checking*. Proceedings of the 6th IEEE Symposium on Logic in Computer Science, Amsterdam, July 1991, pp. 406–415.

[Godefroid & 91b] Godefroid, P. & Wolper, P.: *Using Partial Orders for the Efficient Checking of Deadlock Freedom and Safety Properties*. Proceedings of 3rd Workshop on Computer-Aided Verification 1991, Lecture Notes in Computer Science 575, Springer-Verlag 1992, pp. 332–342.

[Hoare 85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice-Hall 1985, 256 p.

[Holzmann 91] Holzmann, G. J.: *Design and Validation of Computer Protocols*. Prentice-Hall 1991, 500 p.

[Holzmann & 92] Holzmann, G. J.; Godefroid, P. & Pirottin, D.: *Coverage Preserving Reduction Strategies for Reachability Analysis*. 14 p., Participant's Proceedings of the 12th International Symposium on Protocol Specification, Testing and Verification, Lake Buena Vista, Florida, USA, 1992.

[Kaivola & 92] Kaivola, R. & Valmari, A.: *The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic*. Proceedings of CONCUR '92, Lecture Notes in Computer Science 630, Springer-Verlag 1992, pp. 207–221.

[Valmari 88] Valmari, A.: *Error Detection by Reduced Reachability Graph Generation*. Proceedings of the 9th European Workshop on Application and Theory of Petri Nets, Venice, Italy 1988, pp. 95–112.

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

[Valmari 89b] Valmari, A.: *Stubborn Sets for Reduced State Space Generation*. Advances in Petri Nets 1990, Lecture Notes in Computer Science 483, Springer-Verlag 1991, pp. 491–515. (An earlier version appeared in Proceedings of the 10th International Conference on Application and Theory of Petri Nets, Bonn, Germany 1989, Vol. II pp. 1–22.)

[Valmari 90] Valmari, A.: *A Stubborn Attack on State Explosion*. Formal Methods in System Design, 1: 297–322 (1992). (An earlier version appeared in Workshop on Computer-Aided Verification, New Brunswick, NJ, USA 1990.)

[Valmari & 91] Valmari, A. & Tienari, M.: *An Improved Failures Equivalence for Finite State Systems with a Reduction Algorithm*. Protocol Specification, Testing and Verification XI (Proceedings of International IFIP WG 6.1 Symposium, Stockholm, Sweden 1991), North-Holland 1991, pp. 3–18.

[Valmari 92] Valmari, A.: *Alleviating State Explosion during Verification of Behavioural Equivalence*. University of Helsinki, Department of Computer Science, Report A-1992-4, Helsinki 1992, 57 p.