Nordic Journal of Computing

# IMPROVED ON-THE-FLY VERIFICATION WITH TESTERS

TIMO LATVALA
HEIKKI TAURIAINEN
*Laboratory for Theoretical Computer Science, Helsinki University of Technology*
*P.O. Box 5400, FI-02015 HUT, Finland*
{Timo.Latvala,Heikki.Tauriainen}@hut.fi

**Abstract.** We present a new memory efficient algorithm for on-the-fly verification of labelled transition systems (LTSs) with testers. To our knowledge, this is the first thoroughly presented solution for verifying *all* properties specifiable with testers. The algorithm requires four passes of the state space of the composition of the LTS and the tester.

**CR Classification:** F.1.2 [Computation by Abstract Devices]: Modes of Computation — Interactive and Reactive Computation, D.2.4 [Software Engineering]: Software/Program Verification — Formal Methods

**Key words:** Testers, on-the-fly verification, algorithms

## 1. Introduction

Designing reliable and correctly functioning *reactive systems* is a challenging undertaking for developers. Reactive systems have several features which complicate the design process. The systems can be composed of several cooperating components, forcing the developer to deal with the issues of concurrency and nondeterminism. Because reactive systems are non-terminating by nature, this also introduces issues such as fairness which do not have to be taken into consideration for systems exhibiting only finite behaviours. Additionally, the systems usually function in a highly non-deterministic environment.

The class of reactive systems is economically significant as it covers embedded systems such as digital controllers and mobile phones. The market for embedded systems in general is one hundred times as large as for desktop systems [Eggermont 2002]. In other words, there is substantial financial interest in improving development methods for reactive systems. Currently, the preferred method for validation and verification when developing reactive systems is testing. According to several authors [Sanders and Curran 1994, Ferguson and Korel 1996], more than half of the development costs of software projects in general are spent on testing. Testing and debugging systems exhibiting non-determinism is even more difficult due to, e.g., the challenge of repeating faulty executions.

A complementary approach to testing is to perform formal verification on a formal model of the system. One of the simplest models of computation for reactive systems is communicating finite state automata formalised as *labelled transition*

Received 30th September 2004.

*systems* (LTSs). Each component of the system is separately specified as an LTS. The behaviour of the global system is obtained by computing the synchronisation of the LTSs. This forms the so called *state space* of the system.

This simple model is surprisingly powerful and has been used to model systems ranging from telecommunications protocols [Leppänen and Luukkainen 2000] to space craft guidance systems [Havelund *et al.* 2001]. Unfortunately, because the number of states in the state space can be exponential in the number of synchronising processes, proving even simple properties such as deadlock-freedom is PSPACE-complete in the size of the LTSs (see, e.g., Papadimitriou [1994]). Much research has been devoted to alleviating the so called *state explosion problem* (see, e.g., Valmari [1998] for a review).

Simple properties such as the absence of illegal states and deadlocks can be specified as invariants over the state space. These can be checked by inspecting all states. For specifying and verifying more complex properties than deadlocks and illegal states we can use *testers* [Valmari 1993, Hansen *et al.* 2002]. Testers are especially relevant and useful if the semantic model used or the use of a state space reduction method hides actions of the system (see, e.g., Esparza and Heljanko [2000]). They can also be used to check behavioural preorders and equivalences between LTSs such as the CFFD-preorder [Helovuo and Valmari 2000].

Previously, Valmari [1993] has described an algorithm which detects a subset of the properties specified by a tester in one pass of a state space formed by synchronising the tester with the system. Hansen *et al.* [2002] sketched a solution for combining the approach of Holzmann [1991] with the *nested depth-first search* (NDFS) algorithm of Courcoubetis *et al.* [1992] to handle the general verification problem. Even though Hansen *et al.* [2002] claim that the problem can be solved in only three passes of the state space, this is not obvious from their presentation because a generalised NDFS is needed. Our main contribution is a new memory efficient on-the-fly verification algorithm that combines a generalisation of the NDFS with Valmari's algorithm and is able to check any property specified with a tester in at most four passes of the state space.

## 2. Labelled Transition Systems

We model a reactive system as a finite set of synchronising concurrent processes represented as labelled transition systems. Each individual process has a set of states and a set of actions that name the moves between states the process may take during its operation. In this work, we restrict our attention to finite state systems. Our notation and definitions correspond to those of Valmari [1993].

Definition 1. *A* labelled transition system *is a quadruple* $P = (S, \Sigma, \Delta, s_0)$ *where*

- $S$ *is the finite set of* states*,*
- $\Sigma$ *is the finite set of* actions*,*
- $\Delta \subseteq S \times \Sigma \times S$ *is the set of* transitions*, and*
- $s_0 \in S$ *is the initial state.*

The behaviour of an LTS is described by finite and infinite sequences of actions. We define some notation related to describing various types of these sequences; for this purpose, let $\epsilon$, $\Sigma^n$, $\Sigma^*$ and $\Sigma^\omega$ denote the empty string, the set of strings with $n \geq 0$ symbols, the set of finite strings and the set of infinite strings over $\Sigma$, respectively.

Definition 2. *Let $P = (S, \Sigma, \Delta, s_0)$ be an LTS.*

- *For all $s, s' \in S$ and $a \in \Sigma$, $s \xrightarrow{\;a\;} s'$ iff $(s, a, s') \in \Delta$.*
- *For all $n \geq 0$, $s, s' \in S$ and $\sigma = a_1 a_2 \ldots a_n \in \Sigma^n$, $s \xrightarrow{\;\sigma\;} s'$ iff there exist states $s_1, \ldots, s_{n+1} \in S$ such that $s_1 = s$, $s_{n+1} = s'$, and $s_i \xrightarrow{\;a_i\;} s_{i+1}$ holds for all $1 \leq i \leq n$.*
- *For all $n \geq 0$, $s \in S$ and $\sigma \in \Sigma^n$, $s \xrightarrow{\;\sigma\;}$ iff $s \xrightarrow{\;\sigma\;} s'$ holds for some $s' \in S$.*
- *For all $s \in S$ and $\xi = a_1 a_2 \ldots \in \Sigma^\omega$, $s \xrightarrow{\;\xi\;}$ iff there exist infinitely many states $s_1, s_2, \ldots \in S$ such that $s_1 = s$ and $s_i \xrightarrow{\;a_i\;} s_{i+1}$ holds for all $i \geq 1$.*
- *For all $s \in S$, $next(P, s) = \{a \in \Sigma \mid s \xrightarrow{\;a\;}\}$. If $a \in next(P, s)$, we say that the action $a$ is* enabled *in the state $s$ of the LTS $P$.*
- *For all $s \in S$, $reach(P, s) = \{s' \in S \mid s \xrightarrow{\;\sigma\;} s'$ for some $\sigma \in \Sigma^*\}$.*

Communication between individual processes is modelled by synchronising their corresponding LTSs on common actions. In a system consisting of multiple processes, we allow a single LTS to advance independently from one state to another only if it takes an action that is unique to the LTS; if the action is shared by several LTSs, we require that all of these LTSs participate in the same action. Formally, this synchronisation principle is stated as follows.

Definition 3. *Let $P_1 = (S_1, \Sigma_1, \Delta_1, s_{0_1}), \ldots, P_k = (S_k, \Sigma_k, \Delta_k, s_{0_k})$ be LTSs for some $k \geq 1$. The* parallel composition *of $P_1, P_2, \ldots, P_k$ is the LTS $P_1 \| P_2 \| \cdots \| P_k = (S, \Sigma, \Delta, s_0)$, where*

- *$S = S_1 \times \cdots \times S_k$;*
- *$\Sigma = \bigcup_{i=1}^{k} \Sigma_i$;*
- *for all $s = (s_1, s_2, \ldots, s_k)$, $s' = (s'_1, s'_2, \ldots, s'_k) \in S$ and $a \in \Sigma$, $(s, a, s') \in \Delta$ iff, for all $1 \leq i \leq k$,*
  - *if $a \in \Sigma_i$, then $(s_i, a, s'_i) \in \Delta_i$, and*
  - *$s'_i = s_i$ otherwise;*
- *$s_0 = (s_{0_1}, \ldots, s_{0_k})$.*

The properties of action sequences of an LTS $P = (S, \Sigma, \Delta, s_0)$ are often characterised in terms of the occurrence of actions from a designated set $\Sigma_{vis} \subseteq \Sigma$ called the set of *visible actions*. The visible actions also allow us to describe the behaviour of the LTS in a more abstract way by hiding details about actions, whose exact order of occurrence is not considered relevant to the property of interest. We use the following additional notation.

Definition 4. *Let $P = (S, \Sigma, \Delta, s_0)$ be an LTS, and let $\Sigma_{vis} \subseteq \Sigma$.*

- *For all* $(s, a, s') \in \Delta$, $(s, a, s')$ *is a* visible transition *iff* $a \in \Sigma_{vis}$.
- *For all* $\rho \in \Sigma^* \cup \Sigma^\omega$, $vis(\rho) \in \Sigma^*_{vis} \cup \Sigma^\omega_{vis}$ *is the sequence of actions obtained from* $\rho$ *by projecting it onto its visible actions.*
- *For all* $s, s' \in S$ *and* $\sigma \in \Sigma^*_{vis}$, $s \overset{\sigma}{\Longrightarrow} s'$ *iff* $\sigma = vis(\rho)$ *for some* $\rho \in \Sigma^*$ *such that* $s \xrightarrow{\rho} s'$.
- *For all* $s \in S$ *and* $\xi \in \Sigma^*_{vis} \cup \Sigma^\omega_{vis}$, $s \overset{\xi}{\Longrightarrow}$ *iff* $\xi = vis(\rho)$ *for some* $\rho \in \Sigma^* \cup \Sigma^\omega$ *such that* $s \xrightarrow{\rho}$. *The sequence* $\xi$ *is called a* trace.

Valmari [1993] identifies the following properties of an LTS.

Definition 5. *Let* $P = (S, \Sigma, \Delta, s_0)$ *be an LTS, and let* $\Sigma_{vis} \subseteq \Sigma$.

- $tr(P) = \{\sigma \in \Sigma^*_{vis} \mid s_0 \overset{\sigma}{\Longrightarrow}\}$ *is the set of* finite traces *of* $P$.
- $sfail(P) = \{(\sigma, A) \in \Sigma^*_{vis} \times 2^{\Sigma_{vis}} \mid s_0 \overset{\sigma}{\Longrightarrow} s$ *for some* $s \in S$ *such that* $next(P, s) \subseteq \Sigma_{vis} \setminus A\}$ *is the set of* stable failures *of* $P$.
- $divtr(P) = \{\sigma \in \Sigma^*_{vis} \mid \sigma = vis(\rho)$ *for some* $\rho \in \Sigma^\omega$ *such that* $s_0 \xrightarrow{\rho}\}$ *is the set of* divergence traces *of* $P$.
- $inftr(P) = \{\xi \in \Sigma^\omega_{vis} \mid \xi = vis(\rho)$ *for some* $\rho \in \Sigma^\omega$ *such that* $s_0 \xrightarrow{\rho}\}$ *is the set of* infinite traces *of* $P$.

The set $tr(P)$ represents all traces that the LTS $P$ can generate by taking a finite sequence of actions. Each element $(\sigma, A) \in sfail(P)$ determines a finite trace $\sigma$ that leads $P$ into a state in which it can take only visible actions, none of which is, however, included in $A$. For instance, if $(\sigma, \Sigma_{vis}) \in sfail(P)$, then $P$ has a *deadlock*, i.e., a state with no outgoing transitions. The set $divtr(P)$ contains all finite traces leading to a state in which $P$ can enter an infinite computation that contains no visible actions. The set $inftr(P)$ collects all traces with infinitely many visible actions.

## 3. On-the-fly Verification

Specifying and checking for violations of properties required of finite and infinite traces of a given LTS $P_S$ can be accomplished with a special kind of LTS called a *tester* [Valmari 1993]. Intuitively, a tester for $P_S$ describes a collection of traces that correspond to behaviour that $P_S$ should avoid in order to conform to its expected operational requirements. The fact that the tester is itself an LTS leads to an effective method for actually checking (by analysing the parallel composition of the tester and the LTS $P_S$) whether $P_S$ has such undesirable behaviour. In this section, we present an algorithm for the verification of LTSs using testers. This algorithm is a combination of several well-known verification algorithms [Courcoubetis *et al.* 1992, Valmari 1993] and supports detection of violations of the expected requirements "on-the-fly" while building the parallel composition of the tester with the individual components of the LTS $P_S$. Because constructing the composition in full before analysis is potentially very expensive and inefficient in practice due to state explosion, on-the-fly algorithms have the advantage of potentially finding errors even if full verification of the LTS $P_S$ against the tester is impossible within the available resources.

### *3.1 Testers*

Throughout this section, let $P_S = (S_S, \Sigma_S, \Delta_S, s_{0_S})$ be an LTS (possibly obtained as the parallel composition of many synchronising LTSs as described in Section 2) with a set of visible actions $\Sigma_{vis} \subseteq \Sigma_S$. Formally, a tester is defined as follows:

DEFINITION 6. *[Valmari 1993] Let $\Sigma_{vis}$ be the set of visible actions of $P_S$. A* tester *(for $P_S$) is a tuple $T = (P_T, S_R, S_D, S_L, S_\infty)$, where*

- *$P_T = (S_T, \Sigma_T, \Delta_T, s_{0_T})$ is an LTS;*
- *$\Sigma_T = \Sigma_{vis} \cup \{\tau_T\}$, where $\tau_T$ is a new action unique to the tester (i.e., $\tau_T \notin \Sigma_S$);*
- *$S_R \cup S_D \cup S_L \cup S_\infty \subseteq S$;*
- *$\Delta_T$ contains no $\tau_T$-loops, i.e., if there exist $s \in S_T$, $n \geq 1$ and $a_1 a_2 \ldots a_n \in \Sigma^*$ such that $s \xrightarrow{a_1 a_2 \ldots a_n} s$, then $a_i \neq \tau_T$ for some $1 \leq i \leq n$; and*
- *if $(s, \tau_T, s') \in \Delta_T$ for some $s, s' \in S_T$, then $s \notin S_D$.*

*The sets $S_R$ through $S_\infty$ are called* reject states, deadlock monitor states, livelock monitor states, *and* infinite trace monitor states, *respectively.*

Using the definition of a tester, Valmari [1993] gives a classification for the following types of undesirable properties of the LTS $P_S$:

DEFINITION 7. *Let $T = (P_T, S_R, S_D, S_L, S_\infty)$ be a tester for the LTS $P_S$.*

- *$\sigma \in tr(P_S)$ is an* illegal finite trace *iff $s_{0_T} \overset{\sigma'}{\Longrightarrow} s$ holds for some prefix $\sigma'$ of $\sigma$ and a state $s \in S_R$.*
- *$(\sigma, A) \in sfail(P_S)$ is an* illegal stable failure *iff $s_{0_T} \overset{\sigma}{\Longrightarrow} s$ holds for some $s \in S_D$ such that $next(P_T, s) \subseteq A$.*
- *$\sigma \in divtr(P_S)$ is an* illegal divergence trace *iff $s_{0_T} \overset{\sigma}{\Longrightarrow} s$ holds for some $s \in S_L$.*
- *$\xi \in inftr(P_S)$ is an* illegal infinite trace *iff there exist states $s_1, s_2, \ldots \in S_T$ and actions $a_1, a_2, \ldots \in \Sigma_T$ such that $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$, $vis(a_1 a_2 \ldots) = \xi$, and $s_i \in S_\infty$ holds for infinitely many $i \geq 1$.*

Illegal finite traces and illegal stable failures represent violations of properties that have finite traces as counterexamples (generally known as *safety properties*). Illegal finite traces can be used, for example, to model violations of invariants that should hold along all computation paths of the LTS $P_S$. A classic example is a violation of a mutual exclusion condition. Illegal stable failures are suitable for modelling, e.g., deadlock situations in which two synchronising LTSs end up in states where the LTSs can advance only by synchronising with each other on one of their common actions, none of which, however, is enabled for both LTSs.

Illegal divergence traces and illegal infinite traces specified using the sets $S_L$ and $S_\infty$ correspond to violations of *liveness* properties for which the counterexamples are infinite. Illegal divergence traces can be used to model behaviour in which, for example, the LTS $P_S$ ceases to respond to certain events (represented by the visible actions) without actually deadlocking (i.e., a *livelock*). These traces form a subset of all infinite action sequences with only finitely many visible actions.

Illegal infinite traces, which correspond to infinite sequences of visible actions generated by visiting the set $S_\infty$ infinitely often, can be used to represent violations of fairness requirements.

The specification of infinite sequences of states and actions with infinitely many elements from a given set of states or actions is in close relation to the concept of acceptance of infinite words using *Büchi word automata* (see, for example, Thomas [1990]). As a matter of fact, a tester with no reject, deadlock or livelock monitor states and the $\tau_T$ action is structurally identical to a classic Büchi automaton. However, verification with Büchi automata is usually done using a synchronisation principle that forces an automaton to synchronise with every action of the system, regardless of the visibility of the actions. Such behaviour can be simulated easily with the above restricted kind of testers by considering all actions of the system visible. In general, however, distinguishing between visible and non-visible actions affects the class of sequences of visible actions that can be described using testers. For example, Hansen *et al.* [2002] introduce a subclass of testers called *tester automata*, which they show to be expressively equivalent to Büchi automata that recognise stuttering-insensitive languages.

### *3.2 Verification*

The sets $tr(P_S)$, $sfail(P_S)$, $divtr(P_S)$ and $inftr(P_S)$ can be checked for the existence of illegal finite traces, illegal stable failures, illegal divergence traces and illegal infinite traces, respectively, by analysing the parallel composition of a tester and the LTS $P_S$. To ease the description and the analysis of our verification algorithm, we first define the *extended parallel composition* of a tester and an LTS as follows:

Definition 8. *Let $P_T = (S_T, \Sigma_T, \Delta_T, s_{0_T})$ be an LTS associated with a tester for $P_S$. Let $P_T \| P_S = (S, \Sigma, \Delta, s_0)$ be the parallel composition of $P_T$ and $P_S$. The* extended parallel composition *$(P_T\|P_S)^+$ of $P_T$ and $P_S$ is obtained from $P_T\|P_S$ by defining*

$$(P_T\|P_S)^+ = (S \cup \{s_0^+\}, \Sigma, \Delta \cup \{(s_0^+, \tau_T, s_0)\}, s_0^+),$$

*where $s_0^+$ is a new state not included in $S$.*

Thus, we obtain the extended parallel composition of $P_T$ and $P_S$ from $P_T\|P_S$ by taking a new initial state $s_0^+$ for the extended composition and connecting it to the original initial state of $P_T\|P_S$. Because the states of $S$ are internally formed from pairs of states of $P_T$ and $P_S$, we occasionally write $s_0^+$ also as a pair of states $(s_{0_T}^+, s_{0_S}^+)$, where we assume that $s_{0_T}^+$ and $s_{0_S}^+$ are not included in $S_T \cup S_S$.

It is easy to see that $reach(P_T\|P_S, s_0) = reach((P_T\|P_S)^+, s_0^+)\backslash\{s_0^+\}$, and $(P_T\|P_S)^+$ and $P_T\|P_S$ contain the same loops (more formally, for all $n \geq 1$, $s_1, \ldots, s_n \in S \cup \{s_0^+\}$ and $a_1, \ldots, a_n \in \Sigma$, $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \longrightarrow s_n \xrightarrow{a_n} s_1$ holds in $(P_T\|P_S)^+$ iff $s_i \in reach(P_T\|P_S, s_0)$ for all $1 \leq i \leq n$, and $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \longrightarrow s_n \xrightarrow{a_n} s_1$ holds in $P_T\|P_S$). By combining these facts with Theorem 3.3 of Valmari [1993], we can reduce the question on the existence of illegal elements in the sets $tr(P_S)$, $sfail(P_S)$, $divtr(P_S)$ and $inftr(P_S)$ into the following structural properties of the extended parallel composition of $P_T$ and $P_S$.

THEOREM 1. *Let $P_T = (S_T, \Sigma_T, \Delta_T, s_{0_T})$ be an LTS associated with a tester for $P_S$, and let $(P_T \| P_S)^+ = P = (S, \Sigma, \Delta, s_0)$ be the extended parallel composition of $P_T$ and $P_S$.*

- ○ *$tr(P_S)$ contains an illegal finite trace iff there exists a state $s = (s_T, s_S) \in reach(P, s_0)$ such that $s_T \in S_R$.*
- ○ *$sfail(P_S)$ contains an illegal stable failure iff there exists a state $s = (s_T, s_S) \in reach(P, s_0)$ such that $s_T \in S_D$ and $next(P, s) = \emptyset$.*
- ○ *$divtr(P_S)$ contains an illegal divergence trace iff there exists a state $s_T \in S_L$, an integer $n \geq 1$, states $s_{1_S}, \ldots, s_{n_S} \in S_S$ and actions $a_1, \ldots, a_n \in \Sigma_S \setminus \Sigma_{vis}$ such that $(s_T, s_{1_S}) \in reach(P, s_0)$ and $(s_T, s_{1_S}) \xrightarrow{a_1} (s_T, s_{2_S}) \xrightarrow{a_2} \cdots \longrightarrow (s_T, s_{n_S}) \xrightarrow{a_n} (s_T, s_{1_S})$.*
- ○ *$inftr(P_S)$ contains an illegal infinite trace iff there exists a state $s = (s_T, s_S) \in reach(P, s_0)$, an integer $n \geq 1$ and a sequence of actions $\sigma = a_1 a_2 \ldots a_n \in (\Sigma_S \cup \{\tau_T\})^*$ such that $s_T \in S_\infty$, $(s_T, s_S) \xrightarrow{\sigma} (s_T, s_S)$ and $a_i \in \Sigma_{vis}$ for some $1 \leq i \leq n$.*

### *3.3 Algorithms*

The difficulty of detecting errors specified by a tester varies with the type of error. Finding illegal finite traces and illegal stable failures can be accomplished by traversing the state space of the parallel composition. Detecting illegal divergence traces or illegal infinite traces is more challenging.

Valmari [1993] has presented an algorithm which finds illegal divergence traces in one pass of the state space. The algorithm uses a modified depth-first search to find loops in the state space. Given a state in the parallel composition of a tester and an LTS, the algorithm first explores (in depth-first order) all unvisited states reachable from the state without extending the search beyond visible transitions or states in which the tester component is not in a livelock monitor state. The LTS has an illegal divergence trace if the algorithm reaches a state having a transition to a state on the depth-first search stack. The same step is repeated for all other yet unvisited states encountered (but not entered) during the depth-first search until either an illegal divergence trace is found or the entire state space has been explored.

By Theorem 1, detecting illegal infinite traces amounts to finding a reachable loop that visits a state in $S_\infty \times S_S$ and contains at least one visible transition. In other words, we need to find a loop that fulfils two independent conditions on the occurrence of certain states and transitions. It has been suggested [Valmari 1993, Hansen *et al.* 2002] that the classical NDFS algorithm of Courcoubetis *et al.* [1992] for checking the emptiness of the language accepted by a non-deterministic Büchi word automaton can be used for this purpose. However, this algorithm is in its basic form capable of finding loops that satisfy only one (but not necessarily both) of the conditions. The algorithm can nevertheless be easily extended to handle multiple independent conditions (in automata-theoretic terms, *generalised* Büchi acceptance).

The basic NDFS algorithm (see, e.g., Clarke *et al.* [1999] for a simple exposition with pseudocode) works by traversing the state space in depth-first order. When the algorithm backtracks from a state, a new depth-first search is started if the state belongs to a designated set of states in whose occurrence in a loop we are interested (for example, $S_\infty \times S_S$ in our case). Should this second search encounter its own start state (or, more generally, any state in the depth-first search stack of the first search [Holzmann *et al.* 1997]), the state space contains a loop that visits a state from the designated set of states. Even though the second search may apparently have to be restarted from several states, the complete algorithm can be implemented incrementally without traversing the state space more than two times in the worst case [Courcoubetis *et al.* 1992].

In general, if the loops have to satisfy $k \geq 1$ independent conditions, we can reduce them to one condition by modifying the state space [Emerson and Sistla 1984, Courcoubetis *et al.* 1992]. The basic idea is to "unfold" the state space into $k$ interconnected copies such that we always move from a state in the $i^{\text{th}}$ copy of the state space to the $(1+(i \mod k))^{\text{th}}$ copy of the state space whenever our current state or transition fulfils the $i^{\text{th}}$ condition, and remain in the $i^{\text{th}}$ copy otherwise. A second search is now started when we backtrack from a $j^{\text{th}}$-level state associated with the $j^{\text{th}}$ condition for some fixed $1 \leq j \leq k$. By the arrangement of transitions between the copies of the state space, the unfolded state space has a loop fulfilling the $j^{\text{th}}$ condition iff the original state space has a loop that fulfils all of the $k$ conditions. Thus, by running the NDFS algorithm on the unfolded state space, we can detect loops satisfying $k$ independent conditions in at most $2k$ traversals of the original state space.

The key observation to combining most of the advantages of NDFS and Valmari's one-pass algorithm is that the correctness of NDFS is not dependent upon the search order of the second search. In other words, the second searches do not need to be performed in a depth-first manner. This allows us to add illegal divergence trace detection capabilities to the classic NDFS algorithm by using Valmari's one-pass algorithm for the second searches.

Our algorithm can be seen in Fig. 1. The algorithm is invoked by calling the checkForIllegalTraces function using the initial state of the extended parallel composition of a tester LTS with another LTS as a parameter and with the *level* parameter set to zero. This function implements the top-level depth-first search and simultaneously unfolds the state space to detect loops satisfying two independent conditions as described above. The *level* parameter is used for switching between two copies of the state space (lines 8–9). The function also checks for the existence of illegal finite traces and illegal stable failures. A second search is started by calling the checkForIllegalLivenessTraces function on line 12 before backtracking from a state in the set $S_\infty \times S_S$ in the top-level search when $level = 0$. We could alternatively have chosen to start a second search from visible transitions when $level = 1$. However, because we usually expect to see many more visible actions in the parallel composition than states belonging to $S_\infty \times S_S$, our choice for the *level* parameter corresponds to a simple heuristic for reducing the number of visited states.

Regardless of the strategy for starting second searches, we must start a second search also from the initial state of the extended parallel composition to handle the detection of illegal divergence traces that are not reachable from a state in $S_\infty \times S_S$.

The functions checkForIllegalLivenessTraces and advanceFrontier implement the second search. The search extends by advancing a "frontier" of states surrounding the start state of the second search. For each unvisited state in the frontier, the algorithm invokes a depth-first search (lines 19–20) to check whether an illegal divergence trace can be reached from the state (identically to Valmari's algorithm). The depth-first search will not extend beyond visible transitions or transitions that start from a state in which the tester is not in a livelock monitor state. Instead, the target states of these transitions are added to the *frontier* set as new candidates from which to repeat the search for illegal divergence traces. These steps are repeated until an error is found or the frontier becomes empty.

The generalised search for illegal infinite traces is implemented using two additional copies of the state space. The search moves from the first of these copies to the second one whenever encountering a transition labelled with a visible action or the action $\tau_T$ (line 31). If the search thereafter reaches a state that is present in the depth-first search stack of the top-level search (line 25), the algorithm reports the existence of an illegal infinite trace.

The *visited* set is used to keep track of which copies of a state have already been visited. Because the conditions on lines 10, 20 and 34 guarantee that each recursive call to the checkForIllegalTraces or advanceFrontier function always adds a new element to this set, the search always terminates without entering any state more than four times in the worst case. The correctness of the algorithm is established by the following Theorem.

Theorem 2. *Let $(P_T \| P_S)^+ = (S, \Sigma, \Delta, s_0^+)$ be the extended parallel composition of two LTSs $P_T = (S_T, \Sigma_T, \Delta_T, s_{0_T})$ (associated with a tester $(P_T, S_R, S_D, S_L, S_\infty)$) and $P_S = (S_S, \Sigma_S, \Delta_S, s_{0_S})$ (with a set of visible transitions $\Sigma_{vis} \subseteq \Sigma_S$) given as input for the algorithm shown in Fig. 1. The algorithm reports the existence of an illegal finite trace, an illegal stable failure, an illegal divergence trace, or an illegal infinite trace in $P_S$ iff $tr(P_S)$, $sfail(P_S)$, $divtr(P_S)$ or $inftr(P_S)$ contains such a trace, respectively.*

Proof. For simplicity, we assume that the **report** statements do not have the side effect of terminating the execution of the algorithm; in practice, however, this assumption is not needed if we only wish to check $P_S$ for the existence of *some* violation detected by a tester.

*General remarks; illegal finite traces and illegal stable failures* Because the algorithm only adds elements to the (initially empty) set *visited*, it is easy to see that $(s, \ell) \in visited$ holds for some $s \in S$ and $\ell \in \{0, 1\}$ iff the algorithm calls the checkForIllegalTraces function with the parameters $s$ and $\ell$ at some point of its execution. Using this fact, a simple induction shows that the checkForIllegalTraces function is called at least once for each state $s \in reach((P_T \| P_S)^+, s_0^+)$ (and no other states). Therefore, the algorithm can be seen to be both sound and complete for checking $tr(P_S)$ and $sfail(P_S)$ for illegal finite traces and illegal stable failures by Theorem 1 and the conditions on lines 3 and 4.

**Input:** The pair $(s^+_0, 0)$, where $s^+_0 = (s^+_{0_T}, s^+_{0_S})$ is the initial state of the extended parallel composition $(P_T \| P_S)^+ = (S, \Sigma, \Delta, s^+_0)$ of an LTS $P_T = (S_T, \Sigma_T, \Delta_T, s_{0_T})$ (associated with a tester $T = (P_T, S_R, S_D, S_L, S_\infty)$) and an LTS $P_S = (S_S, \Sigma_S, \Delta_S, s_{0_S})$ having a set of visible actions $\Sigma_{vis} \subseteq \Sigma_S$ ($\Sigma_T = \Sigma_{vis} \cup \{\tau_T\}$, $\tau_T \notin \Sigma_S$)

**Output:** A report telling whether the LTS $P_S$ has any illegal finite traces, illegal stable failures, illegal divergence traces or illegal infinite traces.

**Initialise:** *visited* $:= \emptyset$; *frontier* $:= \emptyset$; *trace_prefix* $:= \emptyset$; *divergent_path* $:= \emptyset$

```
checkForIllegalTraces($s \in S$, $level \in \{0, 1\}$)          (* $s = (s_T, s_S) \in (S_T \times S_S) \cup \{(s^+_{0_T}, s^+_{0_S})\}$ *)
    begin
        if ($s_T \in S_R$) then report "$P_S$ has an illegal finite trace"; fi
        if ($s_T \in S_D$ and $next((P_T \| P_S)^+, s) = \emptyset$) then report "$P_S$ has an illegal stable failure"; fi
        $visited := visited \cup \{(s, level)\}$
        $trace\_prefix := trace\_prefix \cup \{(s, level)\}$
        for all $(s, a, s') \in \Delta$ do
            $\ell := level$
            if (($\ell = 0$ and $s_T \in S_\infty$) or ($\ell = 1$ and $a \in \Sigma_{vis} \cup \{\tau_T\}$)) then $\ell := 1 - \ell$; fi
            if ($(s', \ell) \notin visited$) then checkForIllegalTraces($s'$, $\ell$); fi
        od
        if ($level = 0$ and $s_T \in S_\infty \cup \{s^+_{0_T}\}$) then checkForIllegalLivenessTraces($s$); fi
        $trace\_prefix := trace\_prefix \setminus \{(s, level)\}$
    end
```

```
checkForIllegalLivenessTraces($s \in S$)
    begin
        $frontier := \{(s, 2)\}$
        while ($frontier \neq \emptyset$) do
            choose $(s, level) \in frontier$; $frontier := frontier \setminus \{(s, level)\}$
            if ($(s, level) \notin visited$) then advanceFrontier($s, level$); fi
        od
    end

advanceFrontier($s \in S$, $level \in \{2, 3\}$)          (* $s = (s_T, s_S) \in (S_T \times S_S) \cup \{(s^+_{0_T}, s^+_{0_S})\}$ *)
    begin
        if ($level = 3$ and $(\{(s, 0), (s, 1)\} \cap trace\_prefix \neq \emptyset$) then
            report "$P_S$ has an illegal infinite trace"
        fi
        $visited := visited \cup \{(s, level)\}$
        $divergent\_path := divergent\_path \cup \{s\}$
        for all $(s, a, s') \in \Delta$ do
            if ($a \in \Sigma_{vis} \cup \{\tau_T\}$) then $frontier := frontier \cup \{(s', 3)\}$          (* $P_T$ makes a move *)
            else if ($s_T \in S_L$) then begin          (* $P_T$ in livelock monitor state, only $P_S$ makes a move *)
                if ($s' \in divergent\_path$) then report "$P_S$ has an illegal divergence trace"; fi
                if ($(s', level) \notin visited$) then advanceFrontier($s'$, $level$); fi
            end
            else $frontier := frontier \cup \{(s', level)\}$          (* only $P_S$ makes a move *)
            fi
        od
        $divergent\_path := divergent\_path \setminus \{s\}$
    end
```

**Figure 1**: Algorithm for detecting violations of properties specified with a tester

Analogously to the situation above, $(s,\ell) \in \mathit{visited}$ holds for a pair $(s,\ell) \in S \times \{2,3\}$ iff the advanceFrontier function is called with the parameters $s$ and $\ell$. Additionally, calling this function implies that $s \in \mathit{reach}((P_T \| P_S)^+, \hat{s})$ holds for a state $\hat{s} \in S$ from which the algorithm initiates a second search by calling the checkForIllegalLivenessTraces function on line 12 (and therefore $\hat{s}, s \in \mathit{reach}((P_T \| P_S)^+, s_0^+)$ must hold also). On the other hand, it follows by induction on the length of paths starting from $\hat{s}$ that $\{(s',2),(s',3)\} \cap \mathit{visited} \neq \emptyset$ holds for all $s' \in \mathit{reach}((P_T \| P_S)^+, \hat{s})$ upon the completion of this search, when also any second searches possibly initiated before the search starting from $\hat{s}$ are taken into account.

It is straightforward to check that, if $(s,\ell) \in S \times \{0,1\}$ is the pair added to *trace_prefix* on line 6 (or if $s$ is a state added to *divergent_path* on line 29), then $s \in \mathit{reach}((P_T \| P_S)^+, s')$ holds for all $(s',\ell) \in \mathit{trace\_prefix}$ ($s' \in \mathit{divergent\_path}$) in the beginning of each iteration of the loop on line 7 (30) of the algorithm. (For $s, s' \in \mathit{divergent\_path}$, $s$ is actually reachable from $s'$ by taking only transitions labelled with actions not included in $\Sigma_{vis} \cup \{\tau_T\}$, i.e., actions from $\Sigma_S \setminus \Sigma_{vis}$.)

*Illegal divergence traces* If the algorithm reports an illegal divergence trace (line 33), then the above facts and the conditions on lines 30, 32 and 33 imply the existence of a state $s = (s_T, s_S) \in \mathit{reach}((P_T \| P_S)^+, s_0^+) \cap (S_L \times S_S)$ such that $s \xrightarrow{a} s'$ holds for some $a \in \Sigma_S \setminus \Sigma_{vis}$ and $s' \in \mathit{divergent\_path}$. From the above we now see that $s = s_1 \xrightarrow{a_1} \cdots \longrightarrow s_n \xrightarrow{a_n} s_1$ holds for some $n \geq 1$, $a_1, \ldots, a_n \in \Sigma_S \setminus \Sigma_{vis}$ and $s_1, \ldots, s_n \in S$. Furthermore, because $a_i \notin \Sigma_T$ for all $1 \leq i \leq n$, the definition of parallel composition guarantees that all states $s_i$ ($1 \leq i \leq n$) have $s_T \in S_L$ as their $S_T$-component. Therefore $\mathit{divtr}(P_S)$ contains an illegal divergence trace by Theorem 1.

Conversely, if $\mathit{divtr}(P_S)$ contains an illegal divergence trace, then, by Theorem 1, there exist an integer $n \geq 1$, states $s_T \in S_L$, $s_{1_S}, \ldots, s_{n_S} \in S_S$, and actions $a_1, \ldots, a_n \in \Sigma_S \setminus \Sigma_{vis}$ (equivalently, actions $a_1, \ldots, a_n \notin \Sigma_{vis} \cup \{\tau_T\}$) such that $(s_T, s_{1_S}) \in \mathit{reach}((P_T \| P_S)^+, s_0^+)$ holds, and $(s_T, s_{1_S}) \xrightarrow{a_1} (s_T, s_{2_S}) \xrightarrow{a_2} \cdots \longrightarrow (s_T, s_{n_S}) \xrightarrow{a_n} (s_T, s_{1_S})$. By the condition on line 12, the algorithm will call the checkForIllegalLivenessTraces function for the initial state $s_0^+$ of the parallel composition. Because $(s_T, s_{i_S})$ is reachable from $s_0^+$ for all $1 \leq i \leq n$, it now follows by the above discussion that the algorithm calls the advanceFrontier function for all $(s_T, s_{i_S})$ ($1 \leq i \leq n$) before termination; without loss of generality, we can arrange the indices such that $(s_T, s_{1_S})$ is the first state among $(s_T, s_{1_S}), \ldots, (s_T, s_{n_S})$ for which such a call occurs. But then it can be shown that this call results in a recursive call to the same function using the state $(s_T, s_{n_S})$ as a parameter while $(s_T, s_{1_S})$ is still included in the set *divergent_path*. Because $(s_T, s_{n_S}) \xrightarrow{a_n} (s_T, s_{1_S})$, the algorithm reports the existence of an illegal divergence trace.

*Illegal infinite traces* If the algorithm reports an illegal infinite trace (line 26), then the second search (started from a state $\hat{s} \in \mathit{reach}((P_T \| P_S)^+, s_0^+)$) has reached a state $s \in S$ in a call to the advanceFrontier function with parameters $(s,3)$ such that $(s,\ell) \in \mathit{trace\_prefix}$ holds for some $\ell \in \{0,1\}$. The fact that the *frontier* set did not initially contain any pairs from $S \times \{3\}$ in the beginning of the second search can now be used to show that $s$ was actually reached via a state $s' \in \mathit{reach}((P_T \| P_S)^+, \hat{s})$

such that $s' \xrightarrow{a\sigma} s$ holds for some $a \in \Sigma_{vis} \cup \{\tau_T\}$ and $\sigma \in \Sigma^*$. But then also $\hat{s} \xrightarrow{a_1 a_2 \ldots a_n} \hat{s}$ holds for some $n \geq 1$ and $a_1 a_2 \ldots a_n \in \Sigma^n$ such that $a_i = a$ for some $1 \leq i \leq n$. Because $P_T$ contains no $\tau_T$-loops, there actually exists a $1 \leq j \leq n$ such that $a_j \in \Sigma_{vis}$. Furthermore, because $\hat{s} \in S_\infty \times S_S$, it follows from Theorem 1 that *inftr*$(P_S)$ contains an illegal infinite trace.

Assume that *inftr*$(P_S)$ contains an illegal infinite trace. By Theorem 1, there exist an integer $n \geq 1$, states $s_1 \in \mathit{reach}((P_T \| P_S)^+, s_0^+) \cap (S_\infty \times S_S)$, $s_2, \ldots, s_n \in S$ and actions $a_1, \ldots, a_n \in \Sigma_S \cup \{\tau_T\}$ such that $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \longrightarrow s_n \xrightarrow{a_n} s_1$ and $a_i \in \Sigma_{vis}$ for some $1 \leq i \leq n$. It can be shown that the condition on line 12 will hold for some $\hat{s} \in \{s_1, \ldots, s_n\} \cap (S_\infty \times S_S)$ during the execution of the algorithm, and therefore the algorithm will start a second search from $\hat{s}$ by calling the checkForIllegalLivenessTraces function. Without loss of generality, we may fix $\hat{s}$ to be the first such state (in a loop satisfying the above conditions) encountered during the top-level search. In this case, the way in which the state space is unfolded into two levels during the top-level search guarantees that $\{(s_j, 2), (s_j, 3)\} \cap \mathit{visited} = \emptyset$ holds for all $1 \leq j \leq n$ when the second search begins from $\hat{s}$. Because the exact opposite condition holds for all $1 \leq j \leq n$ at the end of this search, it follows that the second search visits (especially) the state $s_i$. The fact that $s_i \xrightarrow{a_i a_{i+1} \ldots a_n} \hat{s}$ holds for the action $a_i \in \Sigma_{vis}$ now implies that the advanceFrontier function will eventually be called also with parameters $(\hat{s}, 3)$ during the second search. Because $\hat{s}$ is the initial state of the second search, it follows that $(\hat{s}, 0) \in \mathit{trace\_prefix}$, and thus the algorithm reports the existence of an illegal infinite trace. □

The memory consumption of the algorithm is fairly conservative (as is common practice we only evaluate the requirement for randomly accessed memory). A standard adaptation of the *hybrid storage* technique of Godefroid and Holzmann [1993] makes it possible to keep track of states visited by the algorithm using a hash table (indexed with the state descriptors) that stores seven bits of memory with each entry in the table. Four of these bits are used for representing a subset of the four “levels” in which the state has been visited, and the remaining three bits are used for tracking state membership in the *trace_prefix* (two bits) and *divergent_path* (one bit) sets. (The two bits used for testing membership in the *trace_prefix* set are not needed if we choose to report an illegal infinite trace only if the second search reaches its own start state instead of any state in the top-level depth-first search stack. Using the extra bits corresponds to the heuristic optimisation suggested by Holzmann *et al.* [1997] to reduce the number of states explored during the second search.) Because the correctness of the algorithm does not actually depend on the uniqueness of the elements in the *frontier* set or the order in which elements are removed from this set, the set can easily be implemented as a stack, which can be stored on sequentially accessed media.

### *3.4 Alternatives for Checking Liveness Properties*

There are a few options for solving the combined problem of detecting both illegal divergence traces and illegal infinite traces. By first running Valmari’s one-pass algorithm and then the standard generalised NDFS algorithm, we obtain an algorithm that traverses the state space five times in the worst case. However, this solution loses the capability of detecting all violations on-the-fly, since running the algorithms separately forces us to explore the state space in full before invoking the NDFS algorithm if no illegal divergence traces are detected. An on-the-fly solution would be to interleave the non-progress cycle detection algorithm of Holzmann [1991] with the generalised NDFS algorithm. The search for non-progress cycles would be performed in a separate copy of the state space. This solution is the closest to ours and was suggested (however, without any implementation details) by Hansen *et al.* [2002].

Another possibility is to map the problem to the acceptance of infinite words by a finite automaton using the *Streett acceptance condition* (see, for example, Thomas [1990]) and use algorithms for checking the emptiness of the language accepted by such an automaton. In general, the Streett acceptance condition for infinite words over a finite alphabet $\Gamma$ is defined by a finite family of acceptance set pairs $\{(L_1, U_1), \ldots, (L_k, U_k)\}$, where $L_i, U_i \subseteq \Gamma$ for all $1 \leq i \leq k$. An infinite word $\gamma_1\gamma_2\gamma_3 \ldots \in \Gamma^\omega$ is *accepted* iff for all $1 \leq i \leq k$, if $\gamma_j \in L_i$ holds for infinitely many $j$, then also $\gamma_\ell \in U_i$ holds for infinitely many $\ell$.

The mapping to Streett acceptance can now be done in the following way. We first take a disjoint copy $S_{vis}$ of the state set $S$ of the extended parallel composition (with initial state $s_0^+ \in S$) of $P_T$ and $P_S$ by introducing for each element $s \in S$ a corresponding (unique) element $s_{vis} \in S_{vis}$. We then redirect all transitions $(s, a, s') \in \Delta$ with $a \in \Sigma_{vis} \cup \{\tau_T\}$ to their corresponding target states $s'_{vis} \in S_{vis}$ and add to $\Delta$ a transition $(s_{vis}, a, s')$ for each (possibly redirected) $(s, a, s') \in \Delta$ with $s \in S$. Let $\Delta'$ be the resulting set of transitions, and let $S' = S \cup S_{vis}$. Intuitively, the LTS $(S', \Sigma, \Delta', s_0^+)$ now contains an infinite path (beginning at $s_0^+$) which visits infinitely many states from $S_{vis}$ iff the LTS $(S, \Sigma, \Delta, s_0^+)$ has an infinite trace with infinitely many visible actions. We also let $S'_L = (S_L \cup \{s_{vis} \mid s \in S_L\}) \times S_S$ and $S'_\infty = (S_\infty \cup \{s_{vis} \mid s \in S_\infty\}) \times S_S$. By identifying the infinite state sequences generated by the LTS $(S', \Sigma, \Delta', s_0^+)$ from its initial state as infinite words over the alphabet $S'$, we can now reduce the question on the existence of illegal divergence traces or illegal infinite traces in the LTS $P_S$ to the property that one of these state sequences satisfies the Streett acceptance condition $\{(S' \setminus S'_L, S_{vis}), (S_{vis}, S'_\infty)\}$.

Algorithms for checking for the existence of state sequences satisfying the Streett acceptance condition [Emerson and Lei 1987, Henzinger and Telle 1996, Latvala and Heljanko 2000] are based on computing the maximal strongly connected components (MSCCs) of the LTS. However, MSCC-based verification algorithms use more memory than NDFS-based algorithms (logarithmic versus constant overhead in the total number of reachable states per state) and they are not suited for on-the-fly verification in the general case, as in the worst case the state space of the system is a single MSCC.

### 3.5 Counterexamples

When an illegal trace is found, reporting the trace is as important as reporting the existence of the trace itself. Simply knowing that an illegal trace exists does not aid the debugging task.

In the standard NDFS algorithm it is very easy to extract a counterexample trace. Unfortunately, our new algorithm does not inherit this feature for all kinds of errors. Since the algorithm unfolds the state space, all traces must be projected to the original state space before they are reported to the user. The following discussion talks about the original state space to make the presentation simpler.

For illegal stable failures and illegal finite traces an error trace is simple to generate. We only need to print the states in the set *trace_prefix* in the order the states were put there. If an explicit depth-first stack is maintained this is easy. We can produce a trace, generated in depth-first order, from the initial state to a violating state. The trace is usually not the shortest trace possible, however.

In the case of an illegal divergence trace, we can only obtain a partial error trace from the data structures. The states in *trace_prefix* can be used to generate a (possibly empty) trace from the initial state to a state $s$, from where a loop of $S_L$-states is reachable. A sequence of non-visible actions forming a loop of $S_L$-states can be extracted from the set *divergent_path*. However, to obtain a complete counterexample trace, a trace from $s$ to a state in *divergent_path* must be found. The complete counterexample starts from the initial state and leads to $s$, continues from $s$ to a state in *divergent_path* and ends with the loop extracted from *divergent_path*.

The last case of an illegal infinite trace is fairly similar to the case of an illegal divergence trace. When the algorithm detects the existence of an illegal infinite trace, *trace_prefix* contains a path from the initial state to a state $s \in S_\infty$, from which a second search was last started. To complete the counterexample, we have to find a loop that contains a visible action. This loop can be built, for example, from the following three segments: a path from $s$ to a state in *divergent_path*, a path extracted from *divergent_path* to a state $s' \in$ *trace_prefix*, and a path from $s'$ to $s$ extracted from *trace_prefix*. By the properties of the algorithm, we can always choose either the path from $s$ to *divergent_path*, or the path from $s'$ to $s$ so that the loop will contain the required visible action.

Completing the illegal divergence traces and the illegal infinite traces is not algorithmically challenging even though it may require an additional pass of the state space. If finding short counterexamples is important, it is possible, e.g., to adapt the counterexample algorithm for Streett automata presented in [Latvala and Heljanko 2000] for this task. An alternative to doing a search in the parallel composition is to store a pointer to an ancestor for each state during the construction of the parallel composition using secondary storage, as suggested by Stern and Dill [1996]. The counterexamples for illegal divergence and illegal infinite traces could easily be completed using this information.

## 4. Discussion and Conclusions

We have presented an on-the-fly algorithm that can verify LTSs w.r.t. all errors specified by testers in four passes of the state space. The memory consumption of the algorithm is also conservative. It uses seven additional bits per state for bookkeeping.

The new algorithm combines most advantages of the NDFS algorithm and Valmari's one-pass algorithm. It is more memory efficient than algorithms based on computing MSCCs. However, if we are not interested in illegal infinite traces, it is preferable to use Valmari's one-pass algorithm, since our algorithm performs a redundant top-level search in this case. The experiments performed by Hansen *et al.* [2002] support this view. Tool implementors may want to implement Valmari's algorithm as a special case.

Because the definition of a tester does not allow $\tau_T$-loops, it follows that each loop in the extended parallel composition of a tester with an LTS either includes at least one visible action, or all actions in the loop are non-visible actions of the LTS otherwise. In the case we wish to consider all loops of non-visible actions illegal (i.e., when $S_L = S_T$), this fact allows us to verify all tester properties in only two passes of the state space of the composition. In this case we can use our algorithm without unfolding the state space during the top-level and second searches. The verification can now be aborted during the second search of the NDFS not only when finding an illegal divergence trace, but also if we encounter a state with a transition to a state on the depth-first search stack of the top-level search. Reaching such a state implies the existence of a loop which in this special case always corresponds to either an illegal infinite trace or an illegal divergence trace. The exact type of the violation needs to be determined separately, however. This can be done by constructing a counterexample as in the case of illegal infinite traces and checking whether the loop in this counterexample contains a visible transition. This optimisation has already been applied to the implementation of the tester verification problem described in Latvala and Mäkelä [2004].

The above optimisation does not apply, however, in the general case where the set of divergence traces is partitioned into non-empty sets of "legal" and illegal divergence traces. In this case the loop detected by finding a path to a state in the *trace_prefix* set during the second search may correspond to one of the "legal" divergence traces, and thus the verification may not be aborted.

Adapting classic constructions from the theory of generalised Büchi automata to the tester verification problem gives an algorithm that traverses the combined state space of the tester and the system four times in the worst case. This upper bound could still possibly be reduced to three worst-case traversals by combining Valmari's one-pass algorithm with a direct emptiness checking algorithm for generalised Büchi automata, such as the one suggested by Tauriainen [2003, Chapter 6]. In fact, it seems likely that three traversals of the state space are the best that can be achieved using NDFS-based algorithms in the worst case: as described in Section 3.3, detecting illegal infinite traces corresponds to solving the language emptiness problem of generalised Büchi automata with two accepting sets, and there are no known NDFS-based algorithms for solving this problem in fewer than three traversals of the automaton in the worst case.

## Acknowledgements

We thank Keijo Heljanko for comments on drafts of this paper. We also gratefully acknowledge the financial support of Helsinki Graduate School in Computer Science and Engineering (HeCSE), the Academy of Finland, project 53695, and the Nokia Foundation (Timo Latvala).

## References

Clarke, E., Grumberg, O., and Peled, D. 1999. *Model Checking*. The MIT Press.

Courcoubetis, C., Vardi, M. Y., Wolper, P., and Yannakakis, M. 1992. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design 1*, 275–288.

Eggermont, L. D. J., Editor. 2002. *Embedded Systems Roadmap 2002*. STW Technology Foundation, Utrecht, The Netherlands.

Emerson, E. A. and Lei, C.-L. 1987. Modalities for Model Checking: Branching Time Logic Strikes Back. *Science of Computer Programming 8*, 3, 275–306.

Emerson, E. A. and Sistla, A. P. 1984. Deciding Full Branching Time Logic. *Information and Control 61*, 3, 175–201.

Esparza, J. and Heljanko, K. 2000. A New Unfolding Approach to LTL Model Checking. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP 2000)*, Volume 1853 of *LNCS*. Springer-Verlag, 475–486.

Ferguson, R. and Korel, B. 1996. The Chaining Approach for Software Test Data Generation. *ACM Transactions on Software Engineering and Methodology 5*, 1, 63–86.

Godefroid, P. and Holzmann, G. J. 1993. On the Verification of Temporal Properties. In *Proceedings of the IFIP TC6/WG6.1 Thirteenth International Symposium on Protocol Specification, Testing and Verification (PSTV 1993)*, Volume C-16 of *IFIP Transactions*. North-Holland, 109–124.

Hansen, H., Penczek, W., and Valmari, A. 2002. Stuttering-Insensitive Automata for On-the-fly Detection of Livelock Properties. In *Proceedings of the 7th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS 2002)*, Volume 66(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier.

Havelund, K., Lowry, M., and Penix, J. 2001. Formal Analysis of a Space-Craft Controller Using SPIN. *IEEE Transactions on Software Engineering 27*, 8, 749–765.

Helovuo, J. and Valmari, A. 2000. Checking for CFFD-Preorder with Tester Processes. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2000)*, Volume 1785 of *LNCS*. Springer-Verlag, 283–298.

Henzinger, M. R. and Telle, J. A. 1996. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996)*, Volume 1097 of *LNCS*. Springer-Verlag, 16–27.

Holzmann, G. J. 1991. *Design and Validation of Computer Protocols*. Prentice-Hall.

Holzmann, G. J., Peled, D., and Yannakakis, M. 1997. On Nested Depth First Search. In *Proceedings of the 2nd* Spin *Workshop*, Volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 23–32.

Latvala, T. and Heljanko, K. 2000. Coping with Strong Fairness. *Fundamenta Informaticae 43*, 1–4, 175–193.

Latvala, T. and Mäkelä, M. 2004. LTL Model Checking for Modular Petri Nets. In *Proceedings of the 25th International Conference on Application and Theory of Petri Nets (ICATPN 2004)*, Volume 3099 of *LNCS*. Springer-Verlag, 298–311.

Leppänen, S. and Luukkainen, M. 2000. Compositional Verification of a Third Generation Mobile Communication Protocol. In *Proceedings of the International Workshop on Distributed System Validation and Verification (DSVV 2000)*. IEEE, E118–E125.

Papadimitriou, C. 1994. *Computational Complexity*. Addision-Wesley.

Sanders, J. and Curran, E. 1994. *Software Quality: A Framework for Success in Software Development and Support*. Addison-Wesley.

Stern, U. and Dill, D. L. 1996. A New Scheme for Memory-Efficient Probabilistic Verification. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV 1996)*, Volume 69 of *IFIP Conference Proceedings*. Kluwer, 333–348.

Tauriainen, H. 2003. On Translating Linear Temporal Logic into Alternating and Nondeterministic

Automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science.

Thomas, W. 1990. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science: Formal Models and Semantics*. Volume B. Elsevier, 133–191.

Valmari, A. 1993. On-the-fly Verification with Stubborn Sets. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993)*, Volume 697 of *LNCS*. Springer-Verlag, 397–408.

Valmari, A. 1998. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*. Volume 1491 of *LNCS*. Springer-Verlag, 429–528.