URL: http://www.elsevier.nl/locate/entcs/volume66.html 16 pages

Stuttering-Insensitive Automata for On-the-fly Detection of Livelock Properties

Henri Hansen [1], Wojciech Penczek [2], Antti Valmari [3]

[1,3] *Tampere University of Technology,*
Institute of Software Systems
PO Box 553, FIN-33101 Tampere, FINLAND

[2] *Institute of Computer Science, PAS*
Ordona 21, 01-237 Warsaw, POLAND

Abstract

The research examines liveness and progress properties of concurrent systems and their on-the-fly verification. An alternative formalism to Büchi automata, called *testing automata*, is developed. The basic idea of testing automata is to observe *changes* in the values of state propositions instead of the values. Therefore, the testing automata are able to accept only stuttering-insensitive languages. Testing automata can accept the same stuttering-insensitive languages as (state-labelled) Büchi automata, and they have at most the same number of states. They are also more often deterministic. Moreover, on-the-fly verification using testing automata can often (but not always) use an algorithm performing only one search in the state space, whereas on-the-fly verification with Büchi automata requires two searches. Experimental results illustrating the benefits of testing automata are presented.

1 Introduction

In this research we examine liveness and progress properties (see e.g. [11, Chapter 4.2]) of concurrent systems and their on-the-fly verification. On-the-fly verification has the significant benefit that the analysis of an erroneus behaviour is possible when only a fragment of the state space has been generated. This is widely known, and is supported by the measurements in Section 5. The most well-known general-purpose algorithm suitable for on-the-fly verification

[1] Email: hansen@cs.tut.fi
[2] Email: penczek@ipipan.waw.pl
[3] Email: ava@cs.tut.fi

is presented in [3]. It is based on a double search of the state space. The property under inspection is often expressed as a Büchi automaton. From the point of view of the algorithm, the local state of the Büchi automaton is a part of the global state, and as such has an impact on the total number of reachable states.

The basis of this research is the observation that a significant proper subset of liveness properties can be verified by an alternative on-the-fly algorithm. The algorithm does only one search in the state space. It is described as Algorithm 3.4 in [17] but we also present it in Section 3.3. It searches for cycles that represent non-progress. (Another algorithm is the one in [8, pp. 235–237], but unlike the algorithm in [17], it requires a double state space search.) When the verified property can be expressed in a form suitable for the single-search algorithm, the algorithm has a tendency to find an error sooner than the algorithm of [3], as the measurements in Section 5 show.

In this research we develop an alternative formalism to Büchi automata, called *testing automata*, which makes it possible to use the algorithm of [17] in many verification tasks. The basic idea of testing automata is to observe *changes* in the values of state propositions instead of the values. Because of this, the testing automata are able to accept only stuttering-insensitive languages. They can accept the same stuttering-insensitive languages as Büchi automata. They never need more states than state-labelled Büchi automata. Deterministic testing automata accept strictly more stuttering-insensitive languages than Büchi automata.

Many verification researchers find limiting oneself to stuttering-insensitive languages not a serious disadvantage. It may even be seen as a benefit [5,10]. For example, the results in [13] would have been easier to derive, if the authors did not have to bother with the fact that even when the language accepted by a Büchi automaton is stuttering-insensitive, individual local states may be sensitive to stuttering. Confining to stuttering-insensitive languages also expands the possibilities to reduce the automaton before use.

In Section 2 Büchi automata and how they are used in verification are presented. In Section 3 the same is done for testing automata. In Section 4, the relationship between Büchi automata and testing automata is explored. Among other things, a construction is given to transform a Büchi automaton that accepts a stuttering-insensitive language into a testing automaton. In Section 5 some experimental results are given illustrating the benefits of the algorithm in [17].

2 Büchi automata

We will use Büchi automata whose states are labelled rather than the transitions. The intended interpretation is that the automaton has a set of propositions whose truth values depend on the state, and the label of a state indicates the propositions that evaluate to True in that state. Translations between state-labelled and transition-labelled Büchi automata are straightforward, see [12].

2.1 Definitions

A *Büchi automaton* is a 6-tuple

$$(S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}})$$

where

- S is a finite set. Its elements are called *states.*
- Π is a finite set. Its elements are called *propositions.*
- val is a function from S to 2^{Π}. Its elements are called *valuations.*
- $\Delta \subseteq S \times S$. Its elements are called *transitions.*
- $\hat{S} \subseteq S$. Its elements are called *initial states.*
- $F_{\mathsf{inf}} \subseteq S$. Its elements are traditionally called acceptance states but to avoid confusion later, we call them *infinite acceptance states.*

From now on, let $\mathcal{B} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}})$ be a Büchi automaton.

A *run* of $\mathcal{B}$ is an infinite sequence $s_0 s_1 s_2 \cdots \in S^{\omega}$ such that

- $s_0 \in \hat{S}$,
- $\forall i : (s_i, s_{i+1}) \in \Delta$.

A run is *accepting* if and only if $s_i \in F_{\mathsf{inf}}$ holds for infinitely many values of i.

The *language* $\mathcal{L}(\mathcal{B})$ accepted by the Büchi automaton is the set of infinite sequences $P_0 P_1 P_2 \cdots$ s.t. there is an accepting run $s_0 s_1 s_2 \cdots$, where $P_i = val(s_i)$ for $i \geq 0$.

We say that a language $\mathcal{L}$ is *stuttering-insensitive* iff $P_0 P_1 P_2 \cdots \in \mathcal{L} \Leftrightarrow P_0^{i_0} P_1^{i_1} P_2^{i_2} \cdots \in \mathcal{L}$ for every $i_0 > 0, i_1 > 0, \ldots$. Here X^i means a string that consists of i copies of X. A Büchi automaton is stuttering-insensitive, iff the language it accepts is so.

As we have seen already, for a Büchi automaton $\mathcal{B}$, the language $\mathcal{L}(\mathcal{B}) \subseteq (2^{\Pi})^{\omega}$. For certain purposes it is useful to extend the Büchi automata formalism so that they can accept languages $\mathcal{L} \subseteq (2^{\Pi})^{\omega} \cup (2^{\Pi})^{*}$. Therefore, we will discuss two additional kinds of acceptance states, $F_{\mathsf{fin}} \subseteq S$ and $F_{\mathsf{dl}} \subseteq S$ called *finite* and *deadlock acceptance* states, respectively. Let

$$(S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$$

be a Büchi automaton extended in this way. A *run* is defined like above, except that now it may also be finite.

An infinite sequence $P_0 P_1 P_2 \cdots$ is *accepted* if at least one of the two conditions below holds:

(i) $\mathcal{B}$ has an infinite run $s_0 s_1 s_2 \cdots$ such that $\forall i : val(s_i) = P_i$, and $s_0 s_1 s_2 \cdots$

is accepting in the above-defined sense (i.e. $s_i \in F_{\mathsf{inf}}$ for infinitely many i).

(ii) $\mathcal{B}$ has a finite run $s_0 s_1 s_2 \cdots s_k$ for some $k \geq 0$ such that $\forall i \leq k : val(s_i) = P_i$, and $s_k \in F_{\mathsf{fin}}$.

A finite sequence $P_0 P_1 P_2 \cdots P_n$ is *accepted* if at least one of the two conditions below hold:

(iii) $\mathcal{B}$ has a finite run $s_0 s_1 s_2 \cdots s_k$, for some $k \leq n$ such that $\forall i \leq k : val(s_i) = P_i$, and $s_k \in F_{\mathsf{fin}}$.

(iv) $\mathcal{B}$ has a finite run $s_0 s_1 s_2 \cdots s_n$ such that $\forall i \leq n : val(s_i) = P_i$ and $s_n \in F_{\mathsf{dl}}$.

It turns out that the set F_{fin} does not increase the accepting power of Büchi automata, but it has other benefits in verification. The F_{fin}-acceptance can find counterexamples at least as fast as other methods and detection of such counterexamples can be trivially integrated to other methods. It is also sometimes easier or more natural to express properties directly using F_{fin} states than first encoding them as LTL formulas.

Theorem 2.1 *For every Büchi automaton with $F_{\mathsf{fin}} \neq \emptyset$ there is a Büchi automaton with $F_{\mathsf{fin}} = \emptyset$ that accepts the same language.*

Proof. Let $\mathcal{B} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$ be a Büchi automaton, with $F_{\mathsf{fin}} \neq \emptyset$. We construct another automaton $\mathcal{B}' = (S', \Pi, val', \Delta', \hat{S}', F'_{\mathsf{inf}}, \emptyset, F'_{\mathsf{dl}})$ that accepts the same language.

In the construction, we create a clique of states that are all both deadlock and infinite acceptance states, redirect all transitions leading to a finite acceptance state into a state of the clique with the same valuation as the finite acceptance state, and remove all finite acceptance states. This does not affect runs that do not visit such states, and any run that does is accepting in both automata by definition.

- $S_1 = S - F_{\mathsf{fin}}$, $S_2 = 2^{\Pi}$, and $S' = S_1 \cup S_2$.
- $\Delta_1 = \Delta \cap (S_1 \times S_1)$, $\Delta_2 = \{\, (s, val(s')) \mid (s, s') \in \Delta \wedge s \in S_1 \wedge s' \in F_{\mathsf{fin}} \,\}$ and $\Delta_3 = S_2 \times S_2$, and $\Delta' = \Delta_1 \cup \Delta_2 \cup \Delta_3$.
- $val'(s) = val(s)$ whenever $s \in S_1$, and $val'(P) = P$ when $P \in S_2$.
- $\hat{S}' = (\hat{S} \cap S_1) \cup \{\, P \mid \exists s \in F_{\mathsf{fin}} \cap \hat{S} : val(s) = P \,\}$.
- $F'_{\mathsf{inf}} = (F_{\mathsf{inf}} \cap S_1) \cup S_2$.
- $F'_{\mathsf{dl}} = (F_{\mathsf{dl}} \cap S_1) \cup S_2$.

□

The F_{dl}, on the other hand, is meaningful only when dealing with finite sequences. In such cases it is necessary.

We say that a Büchi automaton is *deterministic* iff the following holds: $\forall s, s_1, s_2 \in S : ((s, s_1) \in \Delta \wedge (s, s_2) \in \Delta) \Rightarrow (val(s_1) \neq val(s_2) \vee s_1 = s_2)$.

2.2 Verification with Büchi automata

We define a *system* as a tuple $(S_{\mathsf{S}}, \Pi, val_{\mathsf{S}}, \Delta_{\mathsf{S}}, \hat{S}_{\mathsf{S}})$, where S_{S} is a set of states, Π is a set of propositions, $val_{\mathsf{S}} : S_{\mathsf{S}} \longrightarrow 2^{\Pi}$ is a function that assigns to each state of the system a set of propositions, $\Delta_{\mathsf{S}} \subseteq S_{\mathsf{S}} \times S_{\mathsf{S}}$ is a transition relation, and $\hat{S}_{\mathsf{S}} \subseteq S_{\mathsf{S}}$ is a set of initial states.

A Büchi automaton $\mathcal{B} = (S_{\mathsf{B}}, \Pi, val_{\mathsf{B}}, \Delta_{\mathsf{B}}, \hat{S}_{\mathsf{B}}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$, representing the negation of a tested property, is used in combination with the system. We consider the product $System \parallel \mathcal{B} = (S, \text{“}\rightarrow\text{”}, \hat{S})$, where $S = \{(s,t) \mid s \in S_{\mathsf{S}} \wedge t \in S_{\mathsf{B}} \wedge val_{\mathsf{S}}(s) = val_{\mathsf{B}}(t)\}$, “$\rightarrow$” $\subseteq S \times S$ with $(s,t) \rightarrow (s',t')$ iff $(s,s') \in \Delta_{\mathsf{S}} \wedge (t,t') \in \Delta_{\mathsf{B}}$, and $\hat{S} = S \cap (\hat{S}_{\mathsf{S}} \times \hat{S}_{\mathsf{B}})$.

A state $(s,t) \in S$ is called *reachable* iff there are states $(s_0,t_0),\ldots,(s_n,t_n)$ in S such that $(s_0,t_0) \rightarrow (s_1,t_1) \rightarrow \cdots \rightarrow (s_n,t_n) \wedge (s_0,t_0) \in \hat{S} \wedge (s,t) = (s_n,t_n)$.

In on-the-fly verification, we construct only the reachable part of the product starting from the initial states. We call this part the state space. Furthermore, we can stop immediately after a *counterexample* has been established. This counterexample can be of the following three types:

(i) An infinite sequence of states $(s_0,t_0)(s_1,t_1)\cdots$ such that $(s_0,t_0) \in \hat{S}$ and $\forall i \geq 0 : (s_i,t_i) \rightarrow (s_{i+1},t_{i+1})$ and for infinitely many $i \geq 0$: $t_i \in F_{\mathsf{inf}}$. If S is finite, this means in practice that a cycle reachable from an initial state is found where at least one state of F_{inf} occurs.

(ii) A state (s,t) such that it is reachable, $t \in F_{\mathsf{dl}}$, and there is no s' such that $(s,s') \in \Delta_{\mathsf{S}}$. That is, a deadlock of the system is reachable where the testing automaton is in a state of F_{dl}.

(iii) A reachable state (s,t) such that $t \in F_{\mathsf{fin}}$.

Büchi automata that represent the negations of properties can be either directly built by a system designer or obtained automatically from formulas of Linear Time Temporal Logic (LTL) [1, Section 9.4]. There is a challenge to define an algorithm building automata (for LTL formulas) that are as small as possible, see [7,14,4,6].

The methods aimed at finding counterexamples consist of checking the non-emptiness of the product $System \parallel \mathcal{B}$. Counterexamples of types (ii) or (iii) can be found by checking each state that is encountered during a state space search. There are essentially two methods for finding a counterexample of type (i). One way to accomplish this is to construct the strongly connected components of the state space [2] and then to check whether one of the components contains an acceptance state. This method is not well-suited for on-the-fly verification, because strongly connected components often contain many more states than are needed by the counterexample - it is not unusual that a strongly connected component contains all reachable states. The other method consists of two depth-first-searches [3]; the first one determines and orders the reachable acceptance states, while the second one finds out whether any of the reachable acceptance states is an element of a cycle.

2.3 Heuristics for Büchi automata reduction

Reduction of a Büchi automaton means the construction of a smaller Büchi automaton that accepts the same language as the original one. Reduction can save effort by making the state space smaller. Unfortunately, also the opposite may happen. Consider, for instance, a system and a Büchi automaton, each of which consists of just a cycle of three states with $val(s) = \emptyset$, one of which is an initial state. The state space has three states. However, if the Büchi automaton is reduced to a cycle of two states, the state space grows to six states.

Many reductions for Büchi automata can be obtained from reductions in finite automata and process algebras. Such reductions are heuristics, and therefore they usually do not guarantee minimal results. Some of the heuristics work for all Büchi automata (such as the strong bisimulation minimisation), but for stuttering-insensitive Büchi automata there are more efficient heuristics. A detailed discussion of the reductions would be beyond the page limit of this research, but we introduce some superficially. They are used in the examples of this research, although details of the computations are not necessarily shown.

Let $(S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$ be a Büchi automaton. The following heuristics can be used to reduce its size:

- All states and transitions that are not reachable can be discarded. They can be found in linear time with any elementary graph search algorithm such as depth first [2].
- All states and transitions from which no acceptance state is reachable can be removed. This can be done in linear time by conducting a backwards search starting from acceptance states.
- All transitions starting from a finite acceptance state can be removed.
- If an infinite acceptance state is not in any cycle, it can be removed from F_{inf}.

The following heuristics can be used only if the automaton is stuttering-insensitive:

- F_{fin} can be replaced by $\{ s \in S \mid \exists s_0, \ldots, s_n \in S : s_0 = s \wedge \forall i < n : val(s_i) = val(s_{i+1}) \wedge (s_i, s_{i+1}) \in \Delta \wedge s_n \in F_{\mathsf{fin}} \}$, that is, stuttering immediately before entering a finite acceptance state can be ignored. The same applies to F_{dl}.

3 Testing automata

A *testing automaton* is a variant of an extended Büchi automaton that "reads" a sequence in a different way. The important feature of a testing automaton is that it does not detect valuations, but changes of valuations. Consequently, a testing automaton can accept only stuttering-insensitive languages.

3.1 Definitions

A *testing automaton* is a 9-tuple

$$(S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$$

where

- S is a finite set. Its elements are called *states.*
- Π is a finite set. Its elements are called *propositions.*
- $val : \hat{S} \to 2^{\Pi}$. That is, only initial states are given valuations.
- $\Delta \subseteq S \times (2^{\Pi} - \{\emptyset\}) \times S$. Its elements are called *transitions.*
- $\hat{S} \subseteq S$. Its elements are called *initial states.*
- $F_{\mathsf{inf}} \subseteq S$. Its elements are called *infinite acceptance states.*
- $F_{\mathsf{fin}} \subseteq S$. Its elements are called *finite acceptance states.*
- $F_{\mathsf{dl}} \subseteq S$. Its elements are called *deadlock acceptance states.*
- $F_{\mathsf{ll}} \subseteq S$. Its elements are called *livelock acceptance states.*

A variant of testing automata was defined in [17]. There synchronous communication via transition labels was used instead of Π an *val*. Another notion related to stutter-invariant automata, was defined in [5]. The main difference between these two definitions is in the acceptance criteria; stutter-invariant automata use infinite acceptance states only.

From now on, let $\mathcal{T} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$ be a testing automaton. Define $A \oplus B$ as follows: $A \oplus B = (A - B) \cup (B - A)$.

The testing automaton does not make a move for every symbol that it reads. Instead, it moves only when the valuation changes. To discuss this, we define $\sim s_0 P_0 s_1 P_1 \cdots s_n P_n \rightsquigarrow$ iff $s_0 \in \hat{S} \wedge val(s_0) = P_0 \wedge \forall i < n : ((s_i, P_i \oplus P_{i+1}, s_{i+1}) \in \Delta \wedge P_i \neq P_{i+1}) \vee (s_i = s_{i+1} \wedge P_i = P_{i+1})$. For infinite sequences, $\sim s_0 P_0 s_1 P_1 s_2 P_2 \cdots \rightsquigarrow$ is defined analogously.

An infinite sequence $P_0 P_1 P_2 \cdots$ is *accepted* if at least one of the three conditions below holds:

(i) There are $s_0, s_1, s_2, \ldots \in S$ such that
  - $s_i \in F_{\mathsf{inf}}$ for infinitely many i,
  - $\forall i : \exists k > i : P_i \neq P_k$, and
  - $\sim s_0 P_0 s_1 P_1 s_2 P_2 \cdots \rightsquigarrow$.

(ii) There are $s_0, \ldots, s_k \in S$ such that
  - $s_k \in F_{\mathsf{ll}}$,
  - $\forall i \geq k : P_i = P_k$, and
  - $\sim s_0 P_0 s_1 P_1 \cdots s_k P_k \rightsquigarrow$.

(iii) There are $s_0, \ldots, s_k \in S$ such that

- $s_k \in F_{\mathsf{fin}}$, and
- $\sim s_0 P_0 s_1 P_1 \cdots s_k P_k \rightsquigarrow$.

A finite sequence $P_0 P_1 \cdots P_n$ is *accepted* if at least one of the two conditions below holds:

(iv) There are $s_0, s_1, s_2, \ldots, s_n \in S$ such that
- $s_n \in F_{\mathsf{dl}}$, and
- $\sim s_0 P_0 s_1 P_1 \cdots s_n P_n \rightsquigarrow$.

(v) There are $s_0, s_1, s_2, \ldots, s_k \in S$, for some $k \leq n$ such that
- $s_k \in F_{\mathsf{fin}}$, and
- $\sim s_0 P_0 s_1 P_1 \cdots s_k P_k \rightsquigarrow$.

We say that a testing automaton is *deterministic* iff the following holds: $\forall s, s_1, s_2 \in S : \forall P \subseteq \Pi : ((s, P, s_1) \in \Delta \wedge (s, P, s_2) \in \Delta \Rightarrow s_1 = s_2)$.

3.2 Verification with testing automata

We define a *system* as in Section 2.2. A testing automaton $\mathcal{T} = (S_{\mathsf{T}}, \Pi, val_{\mathsf{T}}, \Delta_{\mathsf{T}}, \hat{S}_{\mathsf{T}}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$ is used in combination with a system, and we consider the product *System* $\parallel \mathcal{T} = (S, \text{“}\rightarrow\text{”}, \hat{S})$, where $S = S_{\mathsf{S}} \times S_{\mathsf{T}}$, $\hat{S} = \{ (s,t) \in \hat{S}_{\mathsf{S}} \times \hat{S}_{\mathsf{T}} \mid val_{\mathsf{S}}(s) = val_{\mathsf{T}}(t) \}$, and $(s,t) \rightarrow (s',t')$ iff one of the following holds:

(i) $(s,s') \in \Delta_{\mathsf{S}} \wedge (t, val_{\mathsf{S}}(s) \oplus val_{\mathsf{S}}(s'), t') \in \Delta_{\mathsf{T}}$

(ii) $(s,s') \in \Delta_{\mathsf{S}} \wedge t = t' \wedge val_{\mathsf{S}}(s) = val_{\mathsf{S}}(s')$

In verification with testing automata, a counterexample can be one of the following:

(i) An infinite sequence of states $(s_0,t_0)(s_1,t_1)\cdots$ such that $(s_0,t_0) \in \hat{S}$ and $\forall i : (s_i,t_i) \rightarrow (s_{i+1},t_{i+1})$ and for infinitely many i: $t_i \in F_{\mathsf{inf}} \wedge val(s_i) \neq val(s_{i+1})$. If S is finite, this means in practice that a cycle is found such that it is reachable from an initial state and there is at least one change in proposition values and at least one state (s',t') in the cycle where $t' \in F_{\mathsf{inf}}$.

(ii) An infinite sequence of states $(s_0,t)(s_1,t)\cdots$ such that $\forall i : (s_i,t) \rightarrow (s_{i+1},t)$ and (s_0,t) is reachable and $t \in F_{\mathsf{ll}}$. That is, a cycle consisting of transitions where no propositions change is found where the testing automaton remains in a state $t \in F_{\mathsf{ll}}$.

(iii) A state (s,t) such that it is reachable, $t \in F_{\mathsf{dl}}$, and there is no s' such that $(s,s') \in \Delta_{\mathsf{S}}$. That is, a deadlock of the system is reachable where the testing automaton is in a state of F_{dl}.

(iv) A reachable state (s,t) such that $t \in F_{\mathsf{fin}}$.

3.3 An algorithm for livelock detection

The counterexamples of type (ii) in the previous subsection can be detected with an algorithm originally published in [17]. We present it here only slightly modified for our purpose and call it one-pass algorithm. Notation is the same as in the previous section.

```
ONE-PASS($(S, \text{“}\rightarrow\text{”}, \hat{S})$)
  Work := $\hat{S}$; Found := $\hat{S}$; $\forall (s_S, s_T) \in \hat{S}$ : $colour((s_S, s_T)) := white$
  while Work $\neq \emptyset$
      choose $(s_S, s_T) \in$ Work; Work := Work $- \{(s_S, s_T)\}$
      if $s_T \in F_{ll}$ then LLDET($(s_S, s_T)$)
      else
          for each $(s'_S, s'_T)$ such that $(s_S, s_T) \rightarrow (s'_S, s'_T)$ do
              if $(s'_S, s'_T) \notin$ Found then
                  Work := Work $\cup \{(s'_S, s'_T)\}$
                  Found := Found $\cup \{(s'_S, s'_T)\}$
                  $colour((s'_S, s'_T)) := white$
  end

LLDET($(s_S, s_T)$)
  if $colour((s_S, s_T)) = black$ then return
  $colour((s_S, s_T)) := gray$
  for each $(s'_S, s'_T)$ such that $(s_S, s_T) \rightarrow (s'_S, s'_T)$ do
      if $val(s_S) = val(s'_S)$ then
          if $(s'_S, s'_T) \notin$ Found then
              Found := Found $\cup \{(s'_S, s'_T)\}$; $colour((s'_S, s'_T)) := white$
              LLDET($(s'_S, s'_T)$)
          else if $colour((s'_S, s'_T)) = gray$ then ERROR FOUND!
          else LLDET($(s'_S, s'_T)$)
      else if$(s'_S, s'_T) \notin$ Found then
          Work := Work $\cup \{(s'_S, s'_T)\}$; Found := Found $\cup \{(s'_S, s'_T)\}$
          $colour((s'_S, s'_T)) := white$
  $colour((s_S, s_T)) := black$
  return
```

If Work is a stack, the outer search (one-pass) is effectively a DFS and if Work is a queue, it is a BFS. We do not commit to any particular way of “choosing” the transitions and states to be explored. It does tend to have a significant effect on the way the algorithm behaves and this effect is explored in Section 5.

3.4 Heuristics for testing automata reduction

All the algorithms in Section 2.3 that do not assume stuttering-insensitivity apply. The remaining are superfluous. In addition:

- When we are only interested in infinite sequences, if there is a state that is both an infinite and livelock acceptance state and has a local loop for each $P \in (2^{\Pi} - \emptyset)$, such a state can be converted into a finite acceptance state. If we are interested also in finite sequences, the state must also be a deadlock acceptance state to begin with.

4 Transformation between Büchi automata and testing automata

4.1 Construction of a testing automaton from a Büchi automaton

Theorem 4.1 *If a Büchi automaton is stuttering-insensitive, then there is a testing automaton that accepts precisely the same sequences, and has the same number of states. If the Büchi automaton is deterministic, then the testing automaton is also deterministic.*

Proof. Let $\mathcal{B} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$ be a stuttering-insensitive Büchi automaton. We construct a testing automaton $\mathcal{T} = (S, \Pi, val^T, \Delta^T, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$ by the following:

- $val^T(s) = val(s)$, whenever $s \in \hat{S}$.
- $\Delta^T = \{ (s, P, s') \mid (s, s') \in \Delta \wedge (P = val(s) \oplus val(s')) \wedge P \neq \emptyset \}$.
- $F_{\mathsf{ll}} = \{ s \in S \mid \exists s_0, s_1, \ldots \in S : s_0 = s \wedge \forall i : val(s_i) = val(s_{i+1}) \wedge (s_i, s_{i+1}) \in \Delta \wedge |\{ i \mid s_i \in F_{\mathsf{inf}} \}| = \infty \}$. These states can be detected by only taking into account transitions $(s, s') \in \Delta$ such that $val(s) = val(s')$ while looking for cycles that contain an F_{inf}-state, and then taking all states from which such an F_{inf}-state is reachable via such transitions.

Note first that nondeterminism is not introduced in the construction. To prove that $\mathcal{L}(\mathcal{T}) = \mathcal{L}(\mathcal{B})$, we take any infinite sequence $P_0 P_1 P_2 \cdots \in \mathcal{L}(\mathcal{B})$. Because the language is stuttering-insensitive, we can assume either that $\forall i : P_i \neq P_{i+1}$ or that $\exists k : \forall i < k : P_i \neq P_{i+1} \wedge \forall i \geq k : P_i = P_{i+1}$. We see that by construction, the testing automaton accepts this sequence. In the case of infinite stuttering, the F_{ll} accepts all the appropriate sequences. Finite sequences $P_0 P_1 \cdots P_n \in \mathcal{L}(\mathcal{B})$ are handled in a similar way. Since the testing automaton ignores stuttering, the inclusion in the other direction should be obvious. $\square$

A similar result, formulated for stutter-invariant automata, can be found in [5].

4.2 Construction of a Büchi automaton from a testing automaton

Theorem 4.2 *Any testing automaton has a corresponding Büchi automaton that accepts precisely the same language.*

Proof. Let $\mathcal{T} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$ be a testing automaton. We will construct a stuttering-insensitive Büchi automaton $(S^B, \Pi, val^B, \Delta^B, \hat{S}^B,$

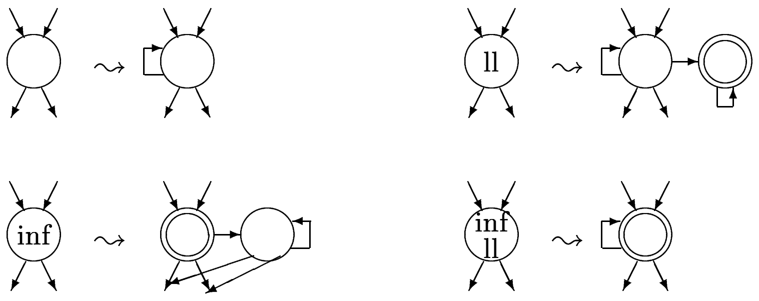

Fig. 1. The construction of a Büchi automaton from a testing automaton

F^B_{inf}, $F^B_{\mathsf{fin}}, F^B_{\mathsf{dl}}$). For clarity, we first construct an intermediate testing automaton $(S', \Pi, val', \Delta', \hat{S}', F'_{\mathsf{inf}}, F'_{\mathsf{fin}}, F'_{\mathsf{dl}}, F'_{\mathsf{ll}})$ such that the values of the propositions in the states are unique.

- Let $S' = S \times 2^{\Pi}$.
- For each $s \in \hat{S}$, put $(s, val(s))$ in $\hat{S}'$.
- $val'((s, P)) = P$ for $s \in \hat{S}'$.
- Δ' is constructed so that whenever $(s, P, s') \in \Delta$, we add $((s, Q), P, (s', Q \oplus P))$ into Δ' for each $Q \in 2^{\Pi}$.
- For each set of acceptance states F_{x}, $F'_{\mathsf{x}} = F_{\mathsf{x}} \times 2^{\Pi}$.

Only the reachable part of this intermediate testing automaton needs to be considered.

The second stage of the construction consists of transforming each state depending on whether it is a member of F_{inf} and/or F_{ll}. These transformations are shown in Figure 1. The *val* function of a state (s, P) is just P. States retain their status as an initial, finite or deadlock acceptance state. The "secondary" states introduced in Figure 1 inherit their *val* values and finite or deadlock acceptance status from their primary states, and are not initial states. □

When an automaton is obtained according to the construction in this proof or the one in Section 4.1, it can often be reduced using the heuristics of Section 2.3 or 3.4.

Theorem 4.3 *There is a deterministic testing automaton such that no deterministic Büchi automaton accepts precisely the same language.*

Proof. Consider the language $\mathcal{L} = (\{P\}|\emptyset)^*\{P\}^\omega$. It is known that it is not accepted by a deterministic Büchi automaton [15]. A deterministic testing automaton accepting this language is shown in Figure 2. □

However, this result turns out coincidental rather than fundamental: there is also a nondeterministic testing automaton such that no deterministic testing automaton accepts precisely the same sequences.

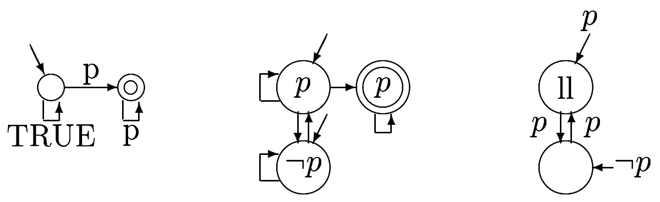

Fig. 2. A transition-labelled and state-labelled Büchi automaton and a testing automaton for the property $\neg\Diamond\Box p$.

5 On-the-fly verification experiments

We compared the algorithm in [3] (from now on just the CVWY-algorithm) to the algorithm in [17], Which we call just one-pass algorithm. For our experiments, we verified the property $\neg\Diamond\Box p$. This property yields a testing automaton that has an ll-state but no inf-states, so the one-pass algorithm applies. Two corresponding Büchi automata and a testing automaton are given in Figure 2. The testing automaton is obtained from the state-labelled Büchi automaton by first using the construction in the proof of Theorem 4.1, and then dropping an unreachable inf-state.

It is easy to compare the performance of the two algorithms when the system works according to the specification, i.e. no counterexample is found. In that case both algorithms construct the whole state space. In addition, the CVWY-algorithm duplicates some of the states.

Theoretical comparison in the case where an error is actually found is much harder, because the effects of the synchronisation of the system and the automaton are complicated, as was discussed in Section 2.3. The goal is in any case to produce as few states as possible before finding the error.

Various aspects must be considered when implementing these algorithms. If the incorrect part of the state space is investigated after all other parts, then, of course, the error is found late. Thus the order in which the transitions of the system are investigated may have a significant effect on the behaviour of the algorithms. This order may be affected by the way in which the system is modelled and represented. The implementation details of the algorithm may also turn out to have a formidable impact on the behavior. For example, depth-first search has a non-recursive implementation where pointers to all successor states of a state are put on the stack in one batch, but these states are not marked as found at this stage. The stack may contain several pointers to the same state, and the state is marked as found when a pointer to it is popped. This implementation scans the transitions in opposite direction from the usual recursive implementation of depth-first search.

In these experiments a total of eight implementations were studied, labelled here with letters from C to J. The meaning of the letters is shown in table 1. "Error First" means that in the search, the acceptance state is searched first. "Forward" and "Backward" refer to the order in which the transitions of the

	one-pass		CVWY	
	BFS	DFS	Error first	Error last
Forward	C	E	G	H
Reverse	D	F	I	J

Table 1
Implementations of algorithms

Alg.	≥ 10000	≥ 1000	≥ 100	total
J	0	5	22	30
I	1	11	28	30
H	0	8	15	30
G	1	13	23	30
F	2	11	16	30
E	0	4	10	30
D	0	5	19	30
C	0	5	20	30

Table 2
Measurement results for the token ring (53856 states)

system are explored.

Ten experiments were made with the famous ten dining philosophers' system. The property was "philosopher i cannot starve in the state where she has one chop stick and is waiting for the other", where i ranged from 1 to 10.

Thirty experiments were made with an artificially 'broken' token-ring system of six servers. A comprehensive description of the token-ring system can be found in [16]. It consists of servers and clients, where the servers are organized in a ring. There is exactly one token, and a server serves a client only when it has the token. A request for the token is passed to the left in the ring and the token is passed to the right. The original token ring guarantees eventual access. The system we study here has such a flaw that the token may sometimes be passed in the wrong direction, introducing the possibility of starvation due to a livelock. Only the servers were included in the model and they had been minimized first. Five possible starvation states were tried for each of the six stations.

Twelve experiments were made with Fischer's mutual exclusion system [9, p. 2] with 12 servers. Each server was monitored for starvation while waiting for access to the critical section.

Alg.	$\geq$ 10000	$\geq$ 1000	$\geq$ 100	total
J	0	4	7	10
I	1	4	10	10
H	0	4	7	10
G	1	4	10	10
F	2	4	8	10
E	2	4	7	10
D	0	0	1	10
C	0	0	1	10

Table 3
Measurement results for dining philosophers (59048 states)

Alg.	$\geq$ 10000	$\geq$ 1000	$\geq$ 100	total
J	2	12	12	12
I	2	12	12	12
H	2	12	12	12
G	2	12	12	12
F	3	6	9	12
E	3	6	9	12
D	0	0	4	12
C	0	0	4	12

Table 4
Measurement results for Fischer's mutex (49153 states)

The number of states generated before detecting the illegal property was recorded. Tables 2, 3 and 4 show how many of these test runs resulted in at least 10 000 states, at least 1000 states and at least 100 states to be generated. It is easy to notice that for all the three systems, the implementations C, D, E, and F are the most effective with C & D outperforming the others. The main advantage in the experimental results is shown for Fischer's mutual exclusion, where the test runs of C & D never generated more than 1000 states, whereas the test runs for G, H, I, and K resulted always in more than 1000 and two times in more than 10 000 states. For dining philosophers C & D generate more than 100 states (but less than 1000) only once, whereas the other implementations generate four times more than 1000 states and at least seven times more than 100 states.

6 Conclusions

In this research we demonstrated with measurements that on-the-fly livelock detection with the algorithm in [17] often outperforms the algorithm in [3], and, to benefit from this observation, developed the notion of a testing automaton. Due to the way a testing automaton observes the system, it is insensitive to stuttering. We gave constructions for transforming a stuttering-insensitive Büchi automaton to a testing automaton that accepts the same language and vice versa, and showed that a testing automaton can be deterministic more often than the Büchi automaton.

Of course, a testing automaton can benefit from the one-pass algorithm only if it contains livelock acceptance states. Even when it does not, Theorem 4.1 guarantees that a minimal testing automaton for a property can have fewer but cannot have more states than a minimal state-labelled Büchi automaton for the same property. However, one must take into account that reducing the number of states in a Büchi or testing automaton does not necessarily reduce the size of the state space – actually the opposite may happen. Because the size of the state space is more important, one should concentrate on it, and not worry too much about the size of the Büchi or testing automaton.

The algorithms in [17] and [3] disagree on the order in which the state space should be investigated, and thus cannot be immediately integrated. This causes a problem for on-the-fly verification with testing automata that contain both livelock and infinite acceptance states. One, albeit not ideal, possibility is to use a triple state space search, where the main and secondary searches would be as in the CVWY-algorithm, and the main search would invoke the third level similarly to the "magic bits" in [8, pp. 235–237].

Testing automata are at their best when Π, the set of propositions, is small. In some cases a testing automaton must remember truth values of propositions in its state, making the number of states grow exponentially in the size of Π. This does not, however, directly make the size of the state space grow, because each state of the system specifies unique values for the propositions, and thus picks only one of the alternative testing automaton states as its pair.

Acknowledgements

The authors would like to thank Maciej Szreter for his help in obtaining the experimental results for Fischer's mutual exclusion algorithm.

References

[1] Clarke, E., Grumberg, O., and Peled, D.: *Model checking*, MIT Press, 1999.

[2] Cormen, T.H., Leiserson, C.E., and Rivest, R.L.: *Introduction to algorithms*, MIT Press,1990.

[3] Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M.: "Memory-efficient algorithms for the verification of temporal properties", *Formal Methods in System Design*, vol. 1., pp. 275–288, 1992.

[4] Etessami, K. and Holzmann, G.: "Optimizing Büchi Automata", in *Proc. of CONCUR'00*, LNCS 1877, 2000.

[5] Etessami, K.: "Stutter-invariant Languages, ω-Automata, and Temporal Logic", in *Proc. of CAV'99*, pp. 236–248, LNCS 1633, 1999.

[6] Gastin, P. and Oddoux, D.: "Fast LTL to Büchi Automata Translation", in *Proc. of CAV'01.*

[7] Gerth, R., Peled, D., Vardi, M., and Wolper, P.: "Simple on-the-fly automatic verification of linear temporal logic", in *Proc. of PSTV'95*, pp. 3–18, 1995.

[8] Holzmann, G: *Design and Validation of Computer Protocols*, Prentice-Hall 1991.

[9] Lamport, L.: "A fast mutual exclusion algorithm", *ACM Transactions on Computer Systems*, 5(1):pp. 1–11, 1987.

[10] Lamport, L.: "What good is temporal logic.". *Proc. IFIP 9th World Congress*, R.E.A. Mason (editor), North-Holland, pp 657 – 668, 1983.

[11] Manna, Z. and Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag 1991.

[12] Peled, D.: "Combining partial order reductions with on-the-fly model-checking", *Formal Methods in System Design 8*, pp. 39–64, 1996.

[13] Peled, D., Valmari, A. and Kokkarinen, I.: "Relaxed Visibility Enhances Partial Order Reduction", *Formal Methods in System Design*, 19, pp. 275–289, 2001.

[14] Somenzi, F. and Bloem, R.: "Efficient Buchi Automata form LTL formulae", in *Proc. of CAV'00*, LNCS 1855, 2000.

[15] Thomas, W.: "Languages, Automata, and Logic", in *Handbook of Formal Languages*, eds. G. Rozenberg and A. Salomaa, Springer-Verlag, pp. 389–455, 1997.

[16] Valmari, A.: "Composition and Abstraction", Cassez, F., Jard, C., Rozoy, B. & Ryan, M. (eds.): *Modelling and Verification of Parallel Processes*, LNCS Tutorials, Lecture Notes in Computer Science 2067, pp. 58–99, Springer-Verlag 2001, .

[17] Valmari, A.: "On-the-fly Verification with Stubborn Sets". *Proc. Computer-Aided Verification (CAV) '93*, Lecture Notes in Computer Science 697, pp. 397–408, Springer-Verlag 1993.