Relaxed Visibility Enhances Partial Order Reduction

Ilkka Kokkarinen[1] and Doron Peled[2] and Antti Valmari[1]

[1] Tampere University of Technology, Software Systems Laboratory,
PO Box 553, FIN-33101 Tampere, FINLAND,
email: {imk,ava}@cs.tut.fi
[2] Bell Laboratories, Lucent Technologies,
700 Mountain Ave., Murray Hill, NJ 07974, USA,
email: doron@research.bell-labs.com

Abstract. State-space explosion is a central problem in the automatic verification (model-checking) of concurrent systems. Partial order reduction is a method that was developed to try to cope with the state-space explosion. Based on the observation that the order of execution of concurrent (independent) atomic actions is in many cases unimportant for the checked property, it allows reducing the state space by exploring fewer execution sequences. However, to be on the safe side, partial order reductions put constraints about commuting the order of atomic actions that may change the value of propositions appearing in the checked specification. In this paper we relax this constraint, allowing a weaker requirement to be imposed, achieving a better reduction. We demonstrate the benefits of our improved reduction with experimental results.

1 Introduction

During the recent years, many practical techniques for the automatic verification of concurrent systems have been developed. One group of such techniques is the *partial order reduction* [3, 5, 10, 11, 13, 14], based on the observation that executing concurrent atomic transitions is commutative. It is then sufficient in many cases to check only a subset of their execution orders instead of all of them. There are two main factors for the effectiveness of such techniques: (a) the amount of commutativity (reflecting independence or concurrency) among atomic program transitions and (b) the number of *visible* atomic transitions, i.e. transitions whose execution may change a predicate in the checked property.

Some attempts with asymmetric or global-state-sensitive dependency relations have been made to reduce the dependency between atomic actions [4, 7, 13] to allow constructing better reduced state spaces. However, only little attention has been made to study the visibility obstacle. In this paper we tackle the problem of relaxing the visibility condition for partial order reduction. Some earlier attempts to make the reduction more sensitive to the checked property eliminate the need of the visibility condition by increasing the dependency according to the checked property. In [5] and [15], the specification was given as an automaton over *operations*, rather than states as in our case. Then any two operations that

can change the state of the property automaton become interdependent. However, this is almost equivalent to the visibility condition (see also [16]). In [10] it is shown that by making a certain fairness assumption and rewriting the property as a Boolean combination of simpler properties, some dependencies between operations that affect the checked property can be eliminated.

We propose a more flexible use of visibility, where the set of visible transitions may diminish during the progress in the construction. We present an improved partial order reduction for model checking that uses this idea. We provide experimental results that show that the new algorithm can achieve substantial improvement over the existing ones.

2 Preliminaries

2.1 Modeling Systems

A *finite state system* P is a triple $\langle S, T, \iota \rangle$, where S is a finite set of *states*, T is a finite set of deterministic *transitions*, and $\iota \in S$ is the *initial state*. For each transition $a \in T$ we associate a partial function $a : S \mapsto S$. A transition a is *enabled* from s, if $a(s)$ is defined. Then, executing a from s results in the state $a(s)$. The set of transitions enabled at a state s is denoted by *enabled*(s).

The full *state-space* of P contains the states reachable from the initial state ι by repeatedly executing the transitions T.

A *transition sequence* is a finite or infinite sequence of transitions $a_0 a_1 a_2 \ldots$ such that there exists a sequence of states $s_0 s_1 s_2 \ldots$ satisfying that (1) $s_0 = \iota$, (2) for each $i \geq 0$, $s_{i+1} = a_i(s_i)$, and (3) the sequence is maximal, namely it is either infinite, or ends with a state s such that *enabled*$(s) = \emptyset$. To save space, we discuss only the case of infinite transition sequences.

A *dependency relation* $D \subseteq T \times T$ is a symmetric and reflexive relation such that if $(a, b) \notin D$, then

- If $a \in$ *enabled*(s), then $b \in$ *enabled*(s) iff $b \in$ *enabled*$(a(s))$.
- If $a, b \in$ *enabled*(s) then $a(b(s)) = b(a(s))$.

Transitions $(a, b) \in D$ are said to be *dependent*, otherwise they are *independent*.

If $\mathcal{P}$ is a finite set of propositional variables, let L be an interpretation function $L : S \mapsto 2^{\mathcal{P}}$, which assigns a boolean value to each proposition in $\mathcal{P}$ for every state in S. We will use the letter ξ to denote the sequence of program states that is induced by a transition sequence. Applying the interpretation function L to each state of ξ, we obtain the *propositional sequence* $L(\xi)$.

Definition 1. We assume that the set of transitions has been divided into *visible* and *invisible* such that if (but not necessarily only if) the execution of a transition from any state of the system can change the value assigned to at least one of the propositions from $\mathcal{P}$, then the transition is visible [14]. □

That is, if there exists a state s and a successor state $s' = a(s)$ such that $L(s) \neq L(s')$, then a is visible. The set of visible transitions is thus some upper approximation of the set of transitions that can modify values of propositions.

An upper approximation is used although it may give less reduction, because the exact set is often too hard to determine.

One of the most popular specification formalisms for concurrent systems is Linear Temporal Logic (LTL) [12]. Its syntax is as follows:

$$\varphi ::= (\varphi_1) \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi_1 \mid \Box\varphi_1 \mid \Diamond\varphi_1 \mid \varphi_1 \mathcal{U} \varphi_2 \mid p$$

where $p \in \mathcal{P}$. We denote a propositional sequence over $2^{\mathcal{P}}$ by σ, and its suffix starting from the ith state (where the first state is numbered 0) by $\sigma^{(i)}$. The boolean operators have their usual interpretations, while the modal operators $\bigcirc$ (*nexttime*), $\Box$ (*always*), $\Diamond$ (*eventually*) and $\mathcal{U}$ (*until*) are interpreted as follows:

- $\sigma \models \bigcirc\varphi$ iff $\sigma^{(1)} \models \varphi$.
- $\sigma \models \Box\varphi$ iff for each $i \geq 0$, $\sigma^{(i)} \models \varphi$.
- $\sigma \models \Diamond\varphi$ iff there exists $i \geq 0$ such that $\sigma^{(i)} \models \varphi$.
- $\sigma \models \varphi \mathcal{U} \psi$ iff $\sigma^{(j)} \models \psi$ for some $j \geq 0$ so that for each $0 \leq i < j$, $\sigma^{(i)} \models \varphi$.

Two propositional sequences w and w' are considered to be *stuttering equivalent*, denoted by $w \equiv w'$, if they differ in at most the number of times the state labeling may adjacently repeat. Every "$\bigcirc$"-free LTL formula obtains the same truth value on any two stuttering equivalent sequences [9], i.e., LTL without nexttime is *stuttering-closed.*

2.2 Partial Order Reduction Algorithms

Many versions have been suggested for partial order reduction algorithms. Most of them have a common basis of doing a search (usually a depth-first search or its variation) on the state space of the checked system (or, in the case of on-the-fly verification, over the combination of program and property states). At each state in the search, only a subset of the successors obtained by executing the enabled transitions is constructed.

Some conditions and algorithms apply to selecting a subset of the enabled transitions from the current state. These conditions must guarantee that the generated state space, albeit smaller than the full state space, preserves the checked property. Developed by different researchers, such subsets adhere to different names: *st*ubborn sets [14], *amp*le sets [11], *per*sistent sets [5] or faithful decompositions [6]. Although their definitions differ, they have much in common. We will call them generically *stamper* sets. For every state s, $stamper(s) \subseteq enabled(s)$, and we call a state s with $stamper(s) = enabled(s)$ *fully expanded.*

It would be futile to try to capture all the subtleties of the various suggested algorithms, some of which take advantage of confining themselves to a more restricted set of properties. We will describe a general algorithm which resembles some of the versions, and can be changed accordingly to capture others. We use the following four conditions.

Non emptiness $stamper(s)$ is empty iff $enabled(s)$ is empty.

Consistency In every finite path of the system that starts from s and does not contain a transition from $stamper(s)$, only transitions independent of the ones in $stamper(s)$ can appear.

Non ignorance Every cycle of the reduced state space contains at least one node t for which $stamper(t) = enabled(t)$.

Visibility If $stamper(s) \neq enabled(s)$, then $stamper(s)$ may not contain transitions that are visible with respect to $\mathcal{P}$.

Let $Seq(P,t)$ be the set of propositional sequences of the concurrent system P, starting from a (not necessarily initial) state t, and $Red(P,t)$ be the corresponding set of propositional sequences of the reduced state space.

Theorem 2. *Let t be a state of the reduced state space. Then $Red(P,t) \subseteq Seq(P,t)$, and for each $\xi \in Seq(P,t)$, there exists a sequence $\xi' \in Red(P,t)$ such that $L(\xi) \equiv L(\xi')$.*

This is the main theorem [11, 14], which holds for many variations of the above rules. We immediately obtain that the reduced state space preserves the correctness of every stuttering-closed property, as follows:

Corollary 3. *For every stuttering-closed property φ, there is a sequence satisfying φ in $Seq(P,t)$ iff there is a sequence satisfying φ in $Red(P,t)$.*

2.3 Translating LTL formulas to automata

In this section we sketch the algorithm presented in [2] for translating an LTL formula φ into a Büchi automaton $\mathcal{A}$.

As a preparatory step, we bring the formula φ into a normal form. First, we push negation inwards, so that only propositional variables can appear negated. To do that, we use LTL equivalences, such as $\neg\Diamond\psi = \Box\neg\psi$. One problem is that pushing negations into until ($\mathcal{U}$) subformulas can explode the size of the formula. For that, we use the operator *release* ($\mathcal{V}$), which is the dual of the operator until, namely $\neg(\mu\,\mathcal{U}\,\eta) = (\neg\mu)\,\mathcal{V}\,(\neg\eta)$. Then we remove the eventuality ($\Diamond$) and always ($\Box$) operators, using the until and release operators and the equivalences $\Diamond\psi = \mathbf{True}\,\mathcal{U}\,\psi$ and $\Box\psi = \mathbf{False}\,\mathcal{V}\,\psi$.

The algorithm uses the following fields for every generated node of $\mathcal{A}$:

id A unique identifier of the node.

incoming The set of edges that are pointed into the node.

new A set of subformulas of the translated formula, which need to hold from the current node and have not yet been processed.

old A set of subformulas as above, which have been processed.

next A set of subformulas of the translated formula, which have to hold for every successor of the current node.

The algorithm starts with a single node, with one incoming edge from a dummy node called *init*. Its field *new* includes the translated formula φ in the above normal form, and the fields *old* and *next* are empty. A list *completed-nodes* is initialized as empty. The algorithm proceeds recursively: for a node x not yet in *completed-nodes*, it moves a subformula η from *new* to *old* [3]. The algorithm

[3] The algorithm can be improved a bit by storing in *old* only subformulas that are atomic propositions, their negations, or of the form $\mu\,\mathcal{U}\,\eta$. The latter are needed for determining the acceptance sets, as explained in the sequel.

then splits the node x into left and right copies while adding subformulas to the fields *new* and *next* according to the following table. The fields *old* and *incoming* retain their previous values in both copies. The algorithm continues recursively with the split copies.

Formula	New left	Next left	New right	Next right
$\mu \mathcal{U} \eta$	$\{\mu\}$	$\{\mu \mathcal{U} \eta\}$	$\{\eta\}$	$\emptyset$
$\mu \mathcal{V} \eta$	$\{\eta\}$	$\{\mu \mathcal{V} \eta\}$	$\{\mu, \eta\}$	$\emptyset$
$\mu \vee \eta$	$\{\mu\}$	$\emptyset$	$\{\eta\}$	$\emptyset$
$\mu \wedge \eta$	$\{\mu, \eta\}$	$\emptyset$	—	—

When there are no more subformulas in the field *new* of the current node x, x is compared against the nodes in the list *completed-nodes*. If there is a node y that agrees with x on the fields *old* and *next*, one adds to the field *incoming* of y the incoming edges of x (hence, one may arrive to the node y from new directions). Otherwise, one adds x to that list and a new node is initiated with a new identifier in *id*, an incoming edge from x, and its *new* field the set of the subformulas in the *next* field of x.

When new nodes can no longer be generated, the set of initial nodes I is identified as those which have an incoming edge from the dummy node *init*. Also an *accepting set* for each subformula of the form $\mu \mathcal{U} \eta$ contains the nodes such that either their *old* field contains the subformula η, or does not contain $\mu \mathcal{U} \eta$. The accepting condition is that of *generalized Büchi* [1], namely, an accepting computation has to traverse *for each accepting set* through at least one of its nodes infinitely often. Each node x is *labeled* by the propositions and negated propositions in its field *old*.

In order to reason about the automaton construction, we use the following notation: let $Next(x)$, $Old(x)$ and $New(x)$ be the subformulas that are in the fields *next*, *old* and *new*, at a specified given point. The conjunction of a set of formulas F will be denoted by $\bigwedge F$, and similarly $\bigvee F$ denotes disjunction. If a node x is repeatedly split to obtain a node y, then x is an *ancestor of* y.

Denote $Form(x) = \bigwedge Old(x) \wedge \bigcirc \bigwedge Next(x)$, at the end of the construction. The above construction has several properties which we exploit for improving the partial order reduction. The following property [2] will be used in the sequel.

Theorem 4. *Let x be a node of the constructed automaton $\mathcal{A}$. Consider the automaton $\mathcal{A}_x$ which is otherwise like $\mathcal{A}$, but has x as its (only) initial state. Then $\mathcal{A}_x$ accepts exactly the sequences satisfying $Form(x)$. Furthermore, for the translated formula φ, $\varphi \leftrightarrow \bigvee_{x \in I} Form(x)$.*

Let $\mathrm{eff}(x)$, the *effective set* of x, be the propositions that appear in the formulas $Old(x) \cup Next(x)$. The following is obvious from the construction:

Lemma 5 Monotonicity. *Let $x \longrightarrow y$ be an edge of $\mathcal{A}$. Then* $\mathrm{eff}(y) \subseteq \mathrm{eff}(x)$.

Lemma 6. *Let x' be an ancestor node of x. If during the translation of a temporal formula into an automaton we have $\psi \in New(x')$, then at the end of the construction $Form(x) \rightarrow \psi$.*

Proof. We prove by induction on the number of splits that for an ancestor x' of y, we have $(\bigwedge New(y) \wedge Form(y)) \rightarrow \bigwedge New(x')$. We then use the fact that at the end of the construction, the field *new* is empty, i.e. **True**. The inductive step of the proof is by cases. For example, let $\psi_1 \vee \psi_2$ be in $New(y)$, and y is split into y_1 and y_2 such that $\psi_1 \in New(y_1)$ and $\psi_2 \in New(y_2)$. Then, $\psi_1 \rightarrow \psi_1 \vee \psi_2$. Hence, $(\bigwedge New(y_1) \wedge Form(y_1)) \rightarrow (\bigwedge New(y) \wedge Form(y))$ and by the inductive hypothesis, $(\bigwedge New(y_1) \wedge Form(y_1)) \rightarrow \bigwedge New(x')$. The proof for node y_2 is similar. □

3 An Improved Algorithm

In Section 2.3, the properties attached to the nodes of the constructed automaton are of the form $\eta = \psi_1 \wedge \bigcirc\psi_2$, with ψ_1, ψ_2 nexttime free. It is not necessarily the case that a property $Form(x)$ attached to an automaton node x is nexttime free. For example, this is not the case when $Old(x) = p$, and $Next(x) = p\,\mathcal{U}\,q$. Then $Form(x)$ allows the sequence $(p,q),(p,q),(\neg p,\neg q),\ldots$, but does not allow the sequence $(p,q),(\neg p,\neg q),\ldots$, which is stuttering equivalent to it.

This is not a problem for the classic on-the-fly stamper set method, because it works at the level of the property. However, to prove the correctness of the new method that will be presented below, it is necessary that each node behaves correctly with respect to the set of transitions that it considers visible. Motivated by this, we investigate closer both the partial order reduction and the automaton construction.

3.1 More Insight About the Reduction

We start by defining a relation between infinite sequences that is stronger than stuttering equivalence.

Definition 7. Let ρ, $\sigma \in \Sigma^\omega$ for some finite alphabet Σ, and let $\alpha \in \Sigma$. Denote $\rho \lhd \sigma$ iff there are γ and γ' such that $\rho = \alpha\gamma$, $\sigma \in \alpha^+\gamma'$, and $\gamma \equiv \gamma'$ (where $\alpha^+ = \{\alpha, \alpha\alpha, \ldots\}$). □

Note that for α, $\beta \in \Sigma$, $\alpha \neq \beta$ and $\gamma \in \Sigma^\omega$, although $\alpha\alpha\beta\gamma \equiv \alpha\beta\gamma$, it does not hold that $\alpha\alpha\beta\gamma \lhd \alpha\beta\gamma$. We can now strengthen the second half of Theorem 2:

Theorem 8. *Let t be a state of the reduced state space. For each $\xi \in Seq(P,t)$, there exists a sequence $\xi' \in Red(P,t)$ such that $L(\xi) \lhd L(\xi')$.*

Proof. Let s be the first state of ξ, and $L(\xi) = \alpha\gamma$. Consider first the case where $stamper(s) = enabled(s)$. Choose a transition τ such that executing τ from s results in s', which is the second state of ξ. Write $\xi = ss'\tilde{\xi}$. Then apply Theorem 2 from s', obtaining a sequence $\tilde{\xi}'$ such that $\gamma = L(s'\tilde{\xi}) \equiv L(s'\tilde{\xi}') = \gamma'$. Then, $\alpha\gamma \lhd \alpha\gamma'$, hence, $L(\xi) \lhd L(ss'\tilde{\xi}')$.

In the second case, we have $stamper(s) \subset enabled(s)$ and all the transitions in $stamper(s)$ are invisible. Now, according to Theorem 2, there is a sequence ξ' such that $L(\xi) \equiv L(\xi')$ constructed from s. Let $\tau \in stamper(s)$ be the first

transition taken to construct ξ'. Since τ is invisible, $L(\xi')$ is of the form $\alpha\alpha\delta$. Since $L(\xi) \equiv L(\xi')$, $\alpha\gamma \equiv \alpha\alpha\delta$. Now, if γ begins with α, let $\gamma' = \alpha\delta$; otherwise, let $\gamma' = \delta^{(i)}$, where i is the smallest value such that $\delta^{(i)}$ does not begin with α. In both cases, by Definition 7, $\alpha\gamma = L(\xi) \angle L(\xi') = \alpha\alpha\delta$. □

Lemma 9. *Let x be a node of the automaton $\mathcal{A}$, and $\rho \angle \sigma$. If $\rho \models Form(x)$, then $\sigma \models Form(x)$.*

Proof. Let $\rho = \alpha\gamma$, with $\alpha \in \Sigma$. Consider first the case where $\sigma = \alpha\gamma'$, with $\gamma \equiv \gamma'$. Then, since both $\bigwedge Old(x)$ and $\bigwedge Next(x)$ are nexttime-free, hence closed under stuttering, we have that $\alpha\gamma \models Form(x)$ iff $\alpha\gamma' \models Form(x)$.

Consider now the case where $\sigma \in \alpha\alpha^+\gamma'$, with $\gamma \equiv \gamma'$. Since any sequence in $\alpha^+\gamma'$ is stuttering equivalent to $\alpha\gamma'$, we can use the above argument to show that $\sigma \models Form(x)$ iff $\alpha\alpha\gamma' \models Form(x)$. Thus, we obtain the claim if we prove $\alpha\alpha\gamma' \models Form(x)$ assuming that $\alpha\gamma \models Form(x)$.

Since $\bigwedge Old(x)$ is nexttime-free, we have that $\alpha\gamma \models Form(x)$ implies that $\alpha\gamma \models \bigwedge Old(x)$, which in turn implies $\alpha\alpha\gamma' \models \bigwedge Old(x)$. Any conjunct in $\bigwedge Next(x)$ can be only of the form $\mu\,\mathcal{U}\,\eta$ or $\mu\,\mathcal{V}\,\eta$. We handle only the latter; the former is similar with η replaced by μ except in $\mu\,\mathcal{U}\,\eta$. When the nexttime-free $\mu\,\mathcal{V}\,\eta$ was added to $Next(x')$, for some ancestor x' of x, η was added to $New(x')$. By Lemma 6, $Form(x) \rightarrow \eta$. Thus, $\alpha\gamma \models \eta$. Since η is nexttime-free, also $\alpha\gamma' \models \eta$. Since $\gamma \models \mu\,\mathcal{V}\,\eta$ and $\mu\,\mathcal{V}\,\eta$ is nexttime-free, $\gamma' \models \mu\,\mathcal{V}\,\eta$. Combining these, we have that $\alpha\gamma' \models \mu\,\mathcal{V}\,\eta$, and by the nexttime-freeness of $\mu\,\mathcal{V}\,\eta$, further that $\alpha\alpha\gamma' \models \mu\,\mathcal{V}\,\eta$. □

Consider the on-the-fly version of the stamper set algorithm. Each state of the state space is of the form $\langle s, x\rangle$, where s is a state of P and x is a state of $\mathcal{A}$. Moreover, $L(s)$ must agree with all propositions and negated propositions in the set $Old(x)$. Transition from $\langle s_1, x_1\rangle$ to $\langle s_2, x_2\rangle$ is only allowed when s_2 is the successor of s_1 under some atomic transition of T, x_2 is a successor of x_1 under the construction of $\mathcal{A}$, and $L(s_2)$ agrees with $Old(x_2)$. Acceptance of a combined state $\langle s, x\rangle$ equals the acceptance of the component x in $\mathcal{A}$. Initial states are of the form $\langle \iota, x\rangle$, where $x \in I$ and $L(\iota)$ agrees with $Old(x)$. The automaton $\mathcal{A}$ is the translation of the *negation* of the checked property φ. A counterexample for φ is obtained by finding a strongly connected component with at least one state for each accepting set of $\mathcal{A}$.[4]

When talking about the on-the-fly algorithm, we will use $stamper(s, x)$ to denote the stamper set used at the joint state with program component s and property component x. Similarly, we denote by $Seq(P, t, x)$ and $Red(P, t, x)$ the set of combined sequences (of system and automaton states) in the combined full and reduced state space, respectively, starting from the node $\langle t, x\rangle$. Such a sequence is accepting iff it passes through each accepting set infinitely often.

We obtain the following on-the-fly version of Theorem 8. The proof is similar to the one in [11], and uses Lemma 9.

[4] This can alternatively be conducted by a multiple depth-first search, with a 'separate' state space for each accepting set, implemented by adding one bit per accepting set [1].

Theorem 10. *Let $\langle t, x\rangle$ be a combined state of the on-the-fly reduced state space. For each accepting sequence ξ in $Seq(P,t,x)$ such that $L(\xi) \models Form(x)$, there exists an accepting sequence ξ' in $Red(P,t,x)$ such that $L(\xi) \angle L(\xi')$ (and hence $L(\xi') \models Form(x)$).*

3.2 An Improved Algorithm

We can now describe the improved algorithm. The only change is to relax the visibility condition. It allows reducing the set of visible transitions, according to the property component x. As the search progresses, the effective propositions in $Form(x)$ may diminish (see Lemma 5), hence less transitions remain visible.

Relative visibility If $stamper(s,x) \neq enabled(s)$, then $stamper(s,x)$ may not contain transitions that are visible w.r.t. the set of propositions $\mathrm{eff}(x)$.

Thus, in the improved algorithm, we start with a set of propositions $\mathcal{P}$, but with each state $\langle s, x\rangle$ in the reduced state space, the set of effective visible transitions is calculated with respect to $\mathrm{eff}(x)$. In Section 4 we show how this can affect the reduction. To prove the improved algorithm correct, we first give two easily provable Lemmas:

Lemma 11. *The improved algorithm finds only correct counterexamples.*

Lemma 12. *Let $\langle s_1, x_1\rangle$, $\langle s_2, x_2\rangle$ be two nodes in the same strongly connected component in the combined reduced state space. Then,* $\mathrm{eff}(x_1) = \mathrm{eff}(x_2)$.

The following theorem is stated with respect to the improved reduction, hence the set of joint sequences from $\langle t, x\rangle$ will be denoted by $Imp\_Red(P,t,x)$.

Theorem 13. *Let $\langle t, x\rangle$ be a combined state of the improved on-the-fly reduced state space. For each accepting sequence ξ in $Seq(P,t,x)$ such that $L(\xi) \models Form(x)$, there exists an accepting sequence ξ' in $Imp\_Red(P,t,x)$ such that $L(\xi) \angle L(\xi')$ (and hence $L(\xi') \models Form(x)$).*

Sketch of proof. By induction on the order of finishing strongly connected components. For the induction basis, consider the last finished strongly connected component. From Lemma 12, there is only one effective set of propositions labeling all the states in that component. Thus, for this component, one can simply use Theorem 10, with visible operations calculated w.r.t. that effective set.

For the inductive step, consider the current component, with edges of the form $\langle s_1, x_1\rangle \longrightarrow \langle s_2, x_2\rangle$, where $\langle s_1, x_1\rangle$ is in the current strongly connected component, and $\langle s_2, x_2\rangle$ is outside it. By the inductive hypothesis, the theorem already holds for the node $\langle s_2, x_2\rangle$. The search of the current component uses visible operations relative to the single effective set of propositions for all of its nodes. It is modified to treat already completed searches from nodes outside the current component, such as $\langle s_2, x_2\rangle$, as oracles about the existence of a desired sequence. Notice that the search from $\langle s_2, x_2\rangle$ is already completed, and has a disjoint set of states from the current component. □

4 Case Study

The purpose of this case study is twofold: to demonstrate that relaxing visibility can yield significant savings, and to give an intuitive idea as to how and when that may be obtained. To achieve the latter goal, we chose a relatively simple example with only the necessary features for illustrating our point. Of course, its results do not generalize to all systems. A problem was that we have not yet implemented our new method and had to do the experiments by playing trickery with an existing tool. We used `ltspar`, a process-algebra-oriented stamper set tool developed by the first author [8]. It computes parallel compositions of synchronously communicating transition systems, and can also use the "transparent" (i.e., not on-the-fly) LTL-preserving stamper set method of Theorems 2 and 8 for computing a reduced parallel composition.

4.1 The Example System

The example concerns the termination of a token-ring system. The system consists of n stations, each of which is capable of sending signals $\mathtt{out}_i$ to the outside world and receivings commands $\mathtt{halt}_i$ for stopping the system. A token circulates in the system, and only the station possessing it can send `out`-signals. The outside world can at any instant of time stop the system by sending some $\mathtt{halt}_i$. The ith station then moves to a halted state and will not pass the token on any more. This will cause the token eventually to stop and the system to terminate.

Each of the stations is modeled by the automaton $\mathrm{STATION}_i$ in Figure 1. In the figure, $\mathtt{tkn}_i$ and $\mathtt{tkn}_{i+1}$ denote the reception of the token from the previous station, and its delivery to the next station (for the nth one, $\mathtt{tkn}_{i+1}$ is replaced by $\mathtt{tkn}_1$). The environment may halt the station at any time by executing $\mathtt{halt}_i$. The label $\mathtt{out}_i$ denotes sending an `out`-signal to the environment. The edge labeled by τ corresponds to the possibility of the station deciding not to send an `out`-signal, although it has received the token. The station 1 initially has the token, so its initial state is 2 while the initial state of all other stations is 1.

The system is modeled by composing n instances of the station graph in parallel. We assume that communication between the stations is synchronous, so the $\mathtt{tkn}_i$-transitions are executed simultaneously by stations i and $i-1$.

4.2 Encoding the Property

Informally, the property that we want to verify of the system is that it stops "soon" after it has been told to stop. More formally, we will verify that for each station i, after *any* station has received a `halt`-signal, the station i will send at most one $\mathtt{out}_i$. We did the experiments in the case where $i = 1$. Except the initial position of the token, the other cases are symmetric.

The following LTL formula encodes the above property. In the formula, $\mathbf{halt}_j$ and $\mathbf{out}_1$ denote the atomic propositions "during its most recent transition, the

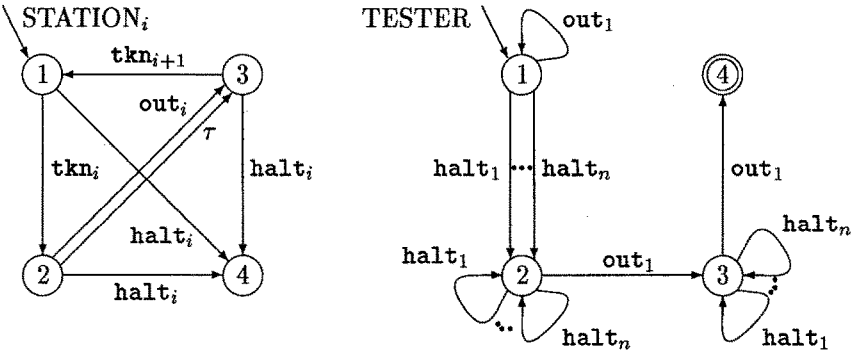

Fig. 1. The ith station process, and the tester process for n stations.

system did read / send the signal $\mathbf{halt}_j$ / $\mathbf{out}_1$".

$$\Box\Big(\Big(\bigvee_{j=1}^{n} \mathbf{halt}_j\Big) \rightarrow \Box\Big(\neg\mathbf{out}_1 \vee (\mathbf{out}_1 \,\mathcal{U}\, \Box\neg\mathbf{out}_1)\Big)\Big)$$

Unfortunately, `ltspar` does not contain support for Büchi automata. Therefore, we used the technique of *tester* processes [15]. We wrote a 4-state tester process, presented on the right in Figure 1, that mimics the negation of the above formula. From the point of view of partial order reductions, testers behave like ordinary processes. Their acceptance condition is different from Büchi automata, but this affects only the "reading" of the result from the reduced state space, not the partial order reduction. Relaxation of visibility affects testers the same way as Büchi automata.[5] Consequently, it was possible to do a first test of our improved on-the-fly algorithm with testers.

To test the performance of the old on-the-fly stamper set method it was sufficient just to add the tester process, declare all actions invisible, and switch the stamper set method on. To test the new method, a small source-level modification was made to `ltspar`, effectively cutting off the dependencies introduced by the tester when the tester is in any other state than the initial state.

4.3 Results

The measurements are shown in Table 1. The sizes of the ordinary state spaces obtained in the absence and presence of the tester process are shown in the columns labeled "full, no tester" and "full with tester". The latter sizes are what the ordinary on-the-fly method (i.e., without stamper set reduction) would yield. The number of states of the "full, no tester" case can be computed theoretically

[5] Actually, with testers the correctness of the relaxation is simpler to prove, because testers talk about the property in terms of transitions instead of propositional variables, and the problem with the nexttime operators needed in the construction of the Büchi automaton does not arise.

n	full, no tester states	full, no tester trans.	reduced, no tester or vis. states	reduced, no tester or vis. trans.	full with tester = old stamper states	full with tester = old stamper trans.	new stamper states	new stamper trans.
2	11	24	9	16	13	25	13	25
3	31	87	17	28	42	106	36	77
4	79	268	28	43	117	367	79	165
5	191	755	42	61	300	1120	151	298
6	447	2010	59	82	731	3151	260	484
7	1023	5145	79	106	1722	8388	414	731
8	2303	12792	102	133	3961	21463	621	1047
9	5119	31095	128	163	8958	53320	889	1440
10	11263	74230	157	196	19959	129463	1226	1918
11	24575	174581	189	232	44022	308644	1640	2489

Table 1. Measurements

relatively easily, yielding $(n+1)2^n - 1$. The "full with tester" state space sizes are more difficult to compute exactly, but they must be somewhat bigger than without the tester, because the tester adds some information of the past behavior to the state and does not restrict the behavior of the system.

The column "old stamper" was obtained in the presence of the tester using the classic on-the-fly stamper set method. The results turned out to be the same as in the column "full with tester", so these two columns were combined into one. This implies that the classic on-the-fly stamper set method gives no reduction in this example. This can be given a theoretical explanation: any station i that is ready to do something is also ready to do a $\mathtt{halt}_i$-transition, and all $\mathtt{halt}_i$-transitions depend on each other as they are all transitions that the tester is ready to make next, so all stamper sets contain all enabled transitions.

To see whether the absence of reduction was due to dependencies within the system, visibility, or both, we constructed also the stamper set state spaces with all transitions invisible and without the tester (remember that the presence of the tester corresponds to making the $\mathtt{out}_1$ and $\mathtt{halt}_i$-transitions visible). The results are shown in the column "reduced, no tester or vis." According to a theoretical analysis, the number of states is $\Theta(n^2)$. Indeed, the measured figures obey the formula $\frac{1}{2}(3n^2 + n + 4)$. The reduction is very good, so the failure of the classic on-the-fly stamper set method is apparently due to the dependencies caused by visibility.

Finally, the column "new stamper" shows the sizes with the new method. Although not as good as in the column "reduced, no tester or vis.", they are still far better than in "old stamper", demonstrating clearly the value of the new method. A theoretical analysis yields $\Theta(n^3)$ states. The measurements match $\frac{1}{6}(8n^3 - 9n^2 + 25n + 6)$ except for $n = 2$.

The results demonstrate that transition dependency caused by the property (i.e., visibility) can significantly hamper the verification of systems, but relaxation of visibility may substantially alleviate this problem.

Acknowledgement The authors would like to thank an insightful discussion of this subject with Amir Pnueli and Pierre Wolper. The work of the first author was partially funded by the Academy of Finland.

References

1. C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, Memory-efficient algorithms for the verification of temporal properties, *Formal Methods in System Design* 1 (1992) 275–288.
2. R. Gerth, D. Peled, M. Vardi, P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, *PSTV95, Protocol Specification Testing and Verification*, 3–18, Chapman & Hall, 1995, Warsaw, Poland.
3. P. Godefroid. Using partial orders to improve automatic verification methods. In Proc. *2nd Workshop on Computer Aided Verification*, LNCS 531, Springer-Verlag, New Brunswick, NJ, 1990, 176–185.
4. P. Godefroid, D. Pirottin, Refining dependencies improves partial order verification methods, *5th Conference on Computer Aided Verification*, Elounda, Greece, LNCS 697, Springer-Verlag, 1993, 438–449.
5. P. Godefroid, P. Wolper, A Partial Approach to Model Checking, *6th Annual IEEE Symposium on Logic in Computer Science*, 1991, Amsterdam, 406–415.
6. S. Katz, D. Peled, Verification of Distributed Programs using Representative Interleaving Sequences, *Distributed Computing* 6 (1992), 107–120.
7. S. Katz, D. Peled, Defining conditional independence using collapses, Theoretical Computer Science 101 (1992), 337–359.
8. I. Kokkarinen, *Reduction of Parallel Labelled Transition Systems with Stubborn Sets*, M. Sc. (Eng.) Thesis (in Finnish), 49 p.
9. L. Lamport, What good is temporal logic, *Information Processing 83*, Elsevier Science Publishers, 1983, 657–668.
10. D. Peled, All from one, one for all, on model-checking using representatives, *5th Conference on Computer Aided Verification*, Elounda, Greece, 1993, LNCS 697, Springer-Verlag, 409–423.
11. D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design* 8 (1996), 39–64.
12. A. Pnueli, The temporal logic of programs, *18th FOCS, IEEE Symposium on Foundation of Computer Science*, 1977, 46–57.
13. A. Valmari, Stubborn sets for reduced state space generation, *10th International Conference on Application and Theory of Petri Nets*, Bonn, Germany, 1989, LNCS 483, Springer-Verlag, 491–515.
14. A. Valmari, A stubborn attack on state explosion. *Formal Methods in System Design*, 1 (1992), 297–322.
15. A. Valmari, On-the-fly Verification with Stubborn Sets, *5th Conference on Computer Aided Verification*, Elounda, Greece, 1993, LNCS 697, Springer-Verlag, 397–408.
16. B. Willems, P. Wolper, Partial-Order Methods for Model Checking: From Linear Time to Branching Time, *11th Annual IEEE Symposium on Logic in Computer Science*, 1996.