A Method for the Verification of Distributed and Synchronized Algorithm

Frederic Magniette, Laurence Pilard and Brigitte Rozoy

Université Paris-Sud, Laboratoire de Recherche en Informatique
Bât 490 - 91405 Orsay Cedex, FRANCE
Fax: 01.69.15.65.86

Frederic Magniette, magniett@lri.fr
Laurence Pilard, pilard@lri.fr, 01.69.15.66.34
Brigitte Rozoy, rozoy@lri.fr, 01.69.15.66.09

Abstract

In the model checking context, the method used to detect stable properties is to construct the synchronized product of the input automata. The problem of such a method is the well known state space explosion. We present a new algorithm that allows to restrict this explosion, by constructing only the states necessary to check the stable properties. The basic idea is to forget the states non relevant to the verification. This is done by agregating the concurent transitions.

KEYWORDS: Model Checking, Verification, Automaton, Synchronized Product

1 Introduction

The main goal of this paper is to derive and to prove an algorithm which use model checking in order to detect stable properties in distributed and synchronised protocols; a stable property, like deadlock, is a property that if it is verified in a state of an execution, then it is also verified in all successors of this state. The main limit of this technique is often due to the excessive size of the state space. Indeed, when the global state space of a system is finite, it is theorically possible to explore the whole of it in order to check properties of the system. In practice, this is often not the case: the global state space is frequently too large to be exhaustively explored. This phenomenon is called the state-explosion problem.

A lot of methods have been proposed to decrease this state-space since ten years, using various system representations as traces, Petri nets, communicating automata, event structures, etc... (Cooper and Marzullo 1991; Godefroid and

Wolper 1991; Burch, Clarke, McMillan, Dill and Hwang 1992; Valmari 1993; Esparza 1994; Jard, Jeron, Jourdan and Rampon 1994; de Souza and de Simone 1994; Peled 1994; Godefroid 1995; Ambroise and Rozoy 1996; Couvreur and Poitrenaud 1996; Abdulla, Annichini, Bensalem, Bouajjani, Habermehl and Lakhnech 1999; Clarke, Grumberg and Peled 1999; Couvreur 1999; Couvreur and Poitrenaud 1999; Esparza and Romer 1999; Esparza and Heljanko 2000; Liu, Stoller and Unnikrishnan 2000; Ambroise 2001; Holzmann 2002). Tools based on these results have been constructed and commercialised (see for example Fernandez, Jard, Jron and Mounier 1992, SPIN 2002...) The known model checking techniques construct a state space associated to all executions of the system, state space that has to be constructed and reduced as effi ciently as possible.

A standard technique is to explore only some executions of the system, using the notion of independence of actions: if two actions A and B are independent in a state, then the state reached after executed AB is the same as the state reached after executed BA. So it is suffi cient to explore the state after the A execution, the state after the B execution, and the state after the BA execution, it is not necessary to explore the state after the AB execution, because it is the same state as the state reached after the BA execution. In this case and for the detection of stable properties, a close idea is used: it is suffi cient to explore only one branch of the execution, i.e. A, then AB, and, as if a stable property is verifi ed in B, then it is still verifi ed in BA $\sim$ AB the analysis of B is useless. So the question is how to determine an appropriate independence relation between actions and to construct only a sub-space of the whole space of the executions without forgetting to construct the space necessary for the verifi cation. There are a lot of techniques that are based on this idea, called partial order methods: sleep sets, persistent sets, borned sets, ...

In the same way, our idea is to exploit the parallelism executing all independent transitions at the same time: in previous example with two independent transitions A and B, we only consruct the state reached after the execution of A and B in parallel, without construct the state reached after the execution of A. For other reasons, a similar idea appears in Devillers, Janicki, Koutny and Lauer (1986), Azema, Michel and Vernadat (1996) and Michel, Vernadat (1997). For us, the reduction factor is about the order of the parallelism degree of the system, the diffi culty always residing in the prediction of execution branches that could appeared in omitted states.

2 Semantics of the system and algorithm idea

We choose here a semantic based on the synchronised product of communicating automata, but the idea developed in our algorithm can be adapted to other contexts as it has been done in Ambroise, Rozoy and Saquet (2003), with a semantic of event structure for asynchronous distributed systems. We could explain the se-

mantic of the system with traces (as Foata or partially commutative monoid). This would have probably been simpler technically, but would necessity have another background.

2.1 Synchronized product of automata

We use the standard definitions, as in Godefroid and Wolper (1991) for instance, and we notice that we only use deterministic automata. In case of non deterministic automata, definitions are slightly different and technically difficult. It's better to first transform them to deterministic one.

Definition 1 *An* automaton *is a set of states of which some are initial, an alphabet and a set of transitions; all states are regarded as final states. The synchronised product of these automata is an automaton; the state space of which is the product of the state space of the input automata, its alphabet is the union of the alphabet of the input automata and a transition*
$S = (s_1, \ldots, s_n) \xrightarrow{a} S' = (s'_1, \ldots, s'_n)$ *exists in the product if and only if:*
(i) for all k such as 'a' is in the alphabet of the k^{th} automaton, $s_k \xrightarrow{a} s'_k$ is a transition of this automaton,
(ii) for all k such as 'a' is not in the alphabet of the k^{th} automaton, $s_k = s'_k$.

To summarize, all individual actions, i.e. actions that appear in only one automaton, can be executed only by this automaton without modifying the rest of the system, while all *common actions* must be executed at the same time by all the automata that contains it. The figure 1 is an illustration of it: the actions a and b are individually executed and their executions are represented by a figure we called a 'square' where $ab \sim ba$, while the actions c, d and e are common. It is well known that the resluting automaton is a *trace automaton* in the partially commutative monoid.

The states of the product automaton will be called *global states* and those of each input automaton, *local states*. Two transitions that are executed over distinct automata will be called *independent*, while two transitions that are executed over at least one common automaton, will be called *dependent*: the execution of one can blocked the execution of the other. For the simplicity of the paper, we only work with product of two automata, extension to more is possible, but technically difficult.

We not only need the states of the product automaton, but also what it did happen before reaching one of these states, i.e. words or equivalent classes of words that describe an execution. We called them configurations. We need this history in order to make efficient backtrack.

Definition 2 *Let A and A' be two automata, and w and w' be two words. We say that $C = (w, w')$ is a* configuration *of the synchronised product of A and A' if*

w and w' belong to the language of the automata A and A' respectively and the couple (w, w') is synchronised over the common transitions, i.e. projection of w and w' on the common alphabet is identical.

In other words, this means that the first automaton could make w while the second made w': (w, w') is a representation of the global execution, without specifying the order in which independent actions are executed. In the example of the figure 1, $(adaadac, bdbbdbbc)$ is a configuration, while $(adaada, bdbbc)$ is not. It is clear that generally, it exists an infinite number of configurations, but these configurations can be easily projected on the finite number of states of the product automaton.

In term of traces, w and w' are projections over each component of the trace C (as Foata representation).

Definition 3 *The state associated to a configuration $C = (w, w')$, called* State(C)*, is the couple of states reached by each automaton, after performing w and w' respectively. We called* Enable(C) *the set of actions of the product automaton that are enabled in the state associated to C.*

Always in the example of the figure 1, for the configuration $C = (ada, bdbb)$, we have *State(C)* = $(1, 2)$ and *Enable(C)* = {a,c}.

2.2 Multi-transitions

Now, we introduce definitions necessary to our algorithm. These definitions are essentially technical, but they allow us to construct a reduced graph $OutAut$ such as the following property is verified : 'If q is a state of the synchronised product and if q is not in $OutAut$, then it exists a state s in $OutAut$ such as a path between q and s exists in the synchronised product.'

Each processus is represented by an automaton and the synchronised product of these automata symbolises the state space graph. In order to detect a stable property, it is sufficient to construct a sub-graph of it such as each not constructed state is covered by a constructed one; if a stable property is verified in a not constructed state, then it will also be verified in all the constructed successors of this state.

We have to construct in an efficient way a sub-graph, using the fact that stable properties are not dependent of the interleaving of independent actions. So, in the whole state space graph, if the transitions $S_0 \xrightarrow{a_1} S_1$,
$S_0 \xrightarrow{a_2} S_2$, ... , $S_0 \xrightarrow{a_n} S_n$ are independent, then we only construct the multi-transition $S_0 \xrightarrow{a_1,a_2,...,a_n} S'$, and we try to choose the set $\{a_1, a_2, \ldots, a_n\}$ as large as possible.

Enabled actions in a configuration will be not executed one by one, because we want to reduce the number of constructed states and transitions. Those actions are

assembled in maximal size packages, so that each package produces only one joint execution.

Definition 4 *Let C be a configuration. In case of two automata,* CoMaxSync(C) *is the set of all sub-sets E of Enable(C) such as: (i) E is composed of independent actions and it is as large as possible, i.e. it doesn't exist in Enable(C) a sub-set E' of independent actions such as $E \subsetneq E'$, or (ii) E is composed by one common action.*

It may be viewed as the subset of actions fired by a maximal subset of processors.

In the example of the figure 1, for the configuration $C_1 = (ada, bdbb)$, we have *CoMaxSync(C_1)* = {{a}, {c}}, for $C_2 = (ada, bdb)$, we have *CoMaxSync(C_2)* = {{a, b}} (neither {a} nor {b} are in *CoMaxSync(C_2)*, because these two sets are not maximal), and finally, for $C_3 = (adaa, bdbb)$, we have *CoMaxSync(C_3)* = {{d}}. Generally, in the case of more practical examples, *CoMaxSync(C)* sets are larger.

Let see now on the example of the figure 2, if this idea seems to be credible: the given input automata allow to fire the maximal *multi-transition* $\{a, b\}$ from the global state $(1, 1)$, then the execution reaches the global state $(3, 2)$. When we execute it, we prevent from executing the common action *c*, because this action is only possible in the global state $(1, 2)$. So, if we don't want to forget this branch, we must make the prevision of this *c* transition. In this way, we introduce the *hope* notion.

2.3 Hope and Cut of a configuration

In a given global state, if only one automaton can execute a common action *c*, then this action may be enable later in the execution, we call this action a *hope* of the considered global state.

In a global state $S = (q, q')$, the common action *c* could be executed, even if a transition $q \xrightarrow{c} s$ exists in the first automaton although transition $q' \xrightarrow{c} s'$ doesn't in the second. Indeed, the transitions $q' \xrightarrow{a_1} q'_1 \xrightarrow{a_2} \ldots \xrightarrow{a_k} q'_k$ and $q'_k \xrightarrow{c} s'$ may exist in the second automaton, then, following the path $(q, q') \xrightarrow{a_1} \ldots \xrightarrow{a_k} (q, q'_k) \xrightarrow{c} (s, s')$, where $a_1, \ldots, a_k$ are independant actions, the common action *c* is executed.

So, in the global state $S = (q, q')$, when a multi-transition $S \xrightarrow{X} R$ is fired, a hope over the common action *c* is transmitted to R, and later R could also transmit this hope. By doing this, we allow us to find a state q'_k in the second automaton, in which the common action *c* is enabled $q'_k \xrightarrow{c} s'$. In this case, a backtrack will be made until the global state (q, q'). We continue the execution by just increasing the

second automaton (single transitions) leading to the (q, q'_k) state.

The hope notion, in a configuration, reflects the existence of a common action enabled only by one automaton, and so not enabled in the configuration, this action can be considered as 'on standby'. A hope is composed by some elements, about input automata and their states, which could allow to fire this common action.

Definition 5 *Let $C = (w, w')$ be a configuration and $S = (s, s')$ be its associated state; a* hope in *C is a 4-tuple (a, num, k, k'), where a is a commom action that is enabled in s or s' but not in both, num is the automaton in which a is enabled, k is the lenght of w and k' is the length of w'.*

Parameters a and num allow detection of omitted global state. Parameters k and k' help to make efficient backtrack.

For the needs of the algorithm procedures, we define *CurrentHope(C)* as the hopes in C and *Hope(C)*, as the hopes transmitted to C. These hopes come from predecessors of C.

The exemple of the figure 3, illustrates the constructed graph using CoMaxSync and Hope techniques, with input automata of the figure 2. In this example, the hope of the state $(1, 1)$ is $[c, 1, -, -]$ in order to express the fact that the first automaton is able to fire the common action c. From $(1, 1)$, we fire the maximal multi-transition $\{a, b\}$, then we reach the global state $(3, 2)$ and we transmit the considered hope to it. Now, the hope of the state $(3, 2)$ is $[c, 2, -, -]$ and the transmitted hopes of it contains $[c, 1, -, -]$. We say that these two hopes are compatibles, because they are both on the same common action c and there is a hope for the first input automaton and for the second. Compatible hopes implie that a global state has been forgotten. This engages a backtrack. We begin to backtrack toward the configuration described in the oldest hope between the two compatible hopes: $[c, 1, 0, 0]$. This hope describe the configuration $(\varepsilon, \varepsilon)$ and the state $(1, 1)$. Then, the other hope describe the different actions we must fire in order to be able to fire the common action c. This hope is $[c, 2, 1, 1]$ and describe the configuration (a, b). Thus the second automaton must fire the action b. The execution of b reaches $(1, 2)$, and finally the execution of c reaches $(2, 3)$.

Moreover, in $(3, 2)$, the common action d is fired, reaching $(4, 4)$, but without any transmission of hope. Indeed, we musn't transmit hopes when a common action is fired, because finding compatible hopes mean that one automaton must go forward, although the other musn't. But if a common action is fired, it is not possible to go forward with only one automaton. In the figure 3, if one automaton doesn't fire the common action d, then this action will not be fired at all.

In order to make efficient backtrack, we need to cut words defining a configuration at a specified length, we call this a *cut*.

Definition 6 *Let $C = (w, w')$ be a configuration and n, n' be two integer; a* cut of C by (n, n')*, called* Cut(C, n, n') *is a couple of words (u, u') prefixes of w and w' respectively such as the lenght of u is n and the length of u' is n'.*

Notice that, if the lengths are badly selected, then a cut might not be a configuration, however the upcoming algorithm will always give us a cut that leads to a configuration.

2.4 Multiple processing of a same global state

We construct only a sub-set of successors of a state. This sub-set is defined by the CoMaxSync of this state and its history (hopes). When we reached a known state, its history may be different than the first time we reach this state. In this case, we must process this state again.

In the state $(4, 4)$ of the figure 3, the execution of the multi-transition $\{a, b\}$ reaches the known state $(1, 1)$ with a new hope $[e, 2, 2, 2]$ that wasn't here the first time we reached this state. We also find two new compatible hopes: $[e, 1, 0, 0]$ and $[e, 2, 2, 2]$. At this point, we backtrack towards the state $(4, 4)$, the execution of a reaches $(1, 4)$ and the execution of e reaches $(5, 5)$.

3 The algorithm

The algorithm takes for the input two automata $\mathcal{A}$ and $\mathcal{A}'$. We call $\mathcal{A} \circ \mathcal{A}'$, the synchronised product of $\mathcal{A}$ and $\mathcal{A}'$. It constructs an automaton $OutAut = (C, T)$, sub-graph of $\mathcal{A} \circ \mathcal{A}'$, such as :
C is a set of configuration of $\mathcal{A} \circ \mathcal{A}'$ and
T is a set of transitions such as $\forall t \in T : s \xrightarrow{t} s', \exists c, c' \in C$ such as
State(c)=$s \wedge$ *State(c)*=$s' \wedge$ it exists a path between s and s' in $\mathcal{A} \circ \mathcal{A}'$.
When a transition is added to $OutAut$, the algorithm may label this transition by cyclic if this adding creates a cycle in $OutAut$.

The algorithm suggested above is essentially composed of three procedures. The main procedure *ConsProdSync* constructs configurations using maximal multi-transitions with *CoMaxSync* (section 2.2), while the second procedure *ConsHope* makes backtrack and initiates the construction of forgotten branches (section 2.3). The third procedure *Traverse* walks across a sub-graph of the constructed graph, only if we reached a known state, for which new transmitted hopes appeared (section 2.4).

3.1 Procedure ConsProdSync(C, $\mathcal{A}$, $\mathcal{A}'$)

This procedure computes *CoMaxSync* for the input configuration C. Then it uses this set to construct successors of C. For each constructed configuration, if its associated state already exists in $OutAut$, the procedure *ConsProdSync* is called, else

the procedure *Traverse* is called. Finally, the procedure computes *CurrentHope* for C and uses it to call *ConsHope*.

INPUT : the current configuration C and the set of input automata $\mathcal{A}$, $\mathcal{A}'$.
OUTPUT : the output automaton $OutAut$ will be modified.

$R \leftarrow CoMaxSync(C)$
$NewHope \leftarrow CurrentHope(C)$
for all X in R **do**
  Construct Configuration $S \leftarrow Concat(C, X)$ *//line A, Concat is the*
  if X is a common transition **then** *//standard concatenation*
    $Hope(S) \leftarrow \emptyset$
  else
    $Hope(S) \leftarrow Hope(C) \cup NewHope$
  if it doesn't exist any S' in $OutAut$ such as $State(S') = State(S)$ **then**
    Add configuration S to $OutAut$ *//line A'*
    Add transition *State(C)* $\xrightarrow{X}$ *State(S)* to $OutAut$ *//line* A'_t
    $ConsProdSync(S, \mathcal{A}, \mathcal{A}')$ *//line callProd2*
  else
    Add cyclic transition *State(C)* $\xrightarrow{X}$ *State(S)* to $OutAut$ *//line* A''_t
    $Traverse(S, \emptyset, \mathcal{A}, \mathcal{A}')$
for all H in $NewHope$ **do**
  $ConsHope(C, H, Hope(C), \mathcal{A}, \mathcal{A}')$ *//line callHope2*

3.2 Procedure ConsHope(C, H, OldHope, $\mathcal{A}$, $\mathcal{A}'$)

This procedure takes for input a current configuration C, an element H of its current hopes, the set $OldHope$ of its transmitted hopes and the input automata $\mathcal{A}$, $\mathcal{A}'$.

It tries to find a set of compatible hopes with H and $OldHope$. If it finds a non empty set, then it constructs configurations that have been forgotten (backtrack) and finally, if it is necessary, it calls *ConsProdSync* with these new constructed configurations.

INPUT : a configuration C, an element H of its set of current hopes, the set $OldHope$ of its transmitted hopes and the set of input automata $\mathcal{A}$, $\mathcal{A}'$.
OUTPUT : the output automaton $OutAut$ will be modified.

let $C = (w, w')$ and $H = (e, aut, n, m)$
$Compatible \leftarrow \{(e, aut', n', m') \in OldHope$ such as $aut' \neq aut\}$ *//line compa*
for all v in $Compatible$ **do**
  Construct Configuration $C_{init} \leftarrow Cut(C, n', m')$
  if $aut = \mathcal{A}$ **then**
    Construct Configuration $C_{tmp} \leftarrow Cut(C, n, m')$
    $SupInit \leftarrow \{v' \subset w'$ such as $\exists u'$ with $(w' = u'.v' \wedge |u'| = m')\}$
  else
    Construct Configuration $C_{tmp} \leftarrow Cut(C, n', m)$

$SupInit \leftarrow \{v \subset w$ such as $\exists u$ with $(w = u.v \wedge |u| = n')\}$
Construct Configuration $S \leftarrow Concat(C_{tmp}, e)$ //line B
$Hope(S) \leftarrow \emptyset$
if it doesn't exist any S' in $OutAut$ such as $State(S') = State(S)$ **then**
Add configuration S to $OutAut$ //line B'
Add transition *State(C_{init})* $\xrightarrow{SupInit.e}$ *State(S)* to $OutAut$ //line B'_t
$ConsProdSync(S, \mathcal{A}, \mathcal{A}')$ //line callProd1
else
Add cyclic transition *State(C_{init})* $\xrightarrow{SupInit.e}$ *State(S)* to $OutAut$ //line B''_t

3.3 Procedure Traverse($C, Tcycle, \mathcal{A}, \mathcal{A}'$)

This procedure is called by *ConsProdSync*, when this one constructs a configuration for which its associated state already appeared in $OutAut$. This procedure traverses a sub-graph of $OutAut$ starting from input configuration, in order to find new compatible hope.

While $OutAut$ can contain cycles, it updates a parameter $Tcyle$, which contains all transitions that are in a cycle and that have already been visited. Doing this way, this procedure terminates.

INPUT : the current configuration C, $Tcyle$ that contains all cyclic transition that have already been visited and the set of input automata $\mathcal{A}$, $\mathcal{A}'$.
OUTPUT : the output automaton $OutAut$ will be modified.

$NewHope \leftarrow CurrentHope(C)$
for all H in $NewHope$ **do**
$ConsHope(C, H, Hope(C), \mathcal{A}, \mathcal{A}')$ //line callHope1
for all transitions T = *State(C)* $\xrightarrow{X}$ *State(S)* in $OutAut \setminus Tcycle$ **do**
if T is a cyclic transition, **then**
$Tcycle \leftarrow Tcycle \cup \{T\}$
if X is not a common transition **then**
Construct Configuration $S \leftarrow Concat(C, X)$
$Hope(S) \leftarrow Hope(S) \cup NewHope$
$Traverse(S, Tcycle, \mathcal{A}, \mathcal{A}')$

3.4 Procedure Main($\mathcal{A}$, $\mathcal{A}'$)

This procedure is the main procedure of the algorithm, it calls *ConsProdSync* with the empty configuration.

INPUT : the set of input automata $\mathcal{A}$, $\mathcal{A}'$.
OUTPUT : the output automaton $OutAut$ will be modified.

Main($\mathcal{A}$, $\mathcal{A}'$) =
Add the configuration $(\varepsilon, \varepsilon)$ in $OutAut$
ConsProdSync($(\varepsilon, \varepsilon), \mathcal{A}, \mathcal{A}'$)

4 Proof of the algorithm

4.1 Few technical elements : Level of a configuration

The notion of level is necessary for the proof of the algorithm. The level of a configuration is the minimum length of the paths, composed by transitions or multi-transitions, that allow us to reach it. In other way, the level of a configuration is the depth of its associated trace, that is to say, the longest string associated to the partial order relation.

Definition 7 *Let $C = (w, w')$ be a configuration with*
$w = u_1 A_1 u_2 A_2 \dots u_n A_n u_{n+1}$ and $w' = u'_1 A_1 u'_2 A_2 \dots u'_n A_n u'_{n+1}$, where A_k are common transitions and u_k, u'_k are independent transitions.
The level of C is level(C) = n + *Max*($|u_1|, |u'_1|$) + ... + *Max*($|u_{n+1}|, |u'_{n+1}|$), *where $|u|$ is the length of the word u.*

We will say that a configuration $C_1 = (u, u')$ is included in a configuration C_2 iff it exists a couple (w, w') such as $C_2 = (u.w, u'.w')$.

When we don't make backtrack or construct forgotten branches, we traverse the graph using multi-transitions, and in that case, we only construct configurations that we will call *full configurations*.

A configuration is full iff it is not included in any other configuration having the same level. In term of traces, a configuration is full iff its associated trace isn't extended by trace of the same depth.

Definition 8 *Let $\mathcal{A} = (Q, Q_0, V, T)$ and $\mathcal{A}' = (Q', Q'_0, V', T')$ be two automata, and $\mathcal{A} \circ \mathcal{A}'$ be the synchronised product of these automata.*
Let $C = (u_1 A_1 u_2 A_2 \dots u_n A_n u_{n+1}, u'_1 A_1 u'_2 A_2 \dots u'_n A_n u'_{n+1})$ be a configuration of $\mathcal{A} \circ \mathcal{A}'$. C is full *iff*
$|u_{n+1}| = |u'_{n+1}|$,
or ($|u_{n+1}| < |u'_{n+1}|$) and starting from C, the automaton $\mathcal{A}$ does not allow to fire any independent action,
or ($|u_{n+1}| > |u'_{n+1}|$) and starting from C, the automaton $\mathcal{A}'$ does not allow to fire any independent action.

4.2 Proof

In this section, in order to prove that our algorithm is correct, we will prove the following theorem:

Theorem 1 *Let $\mathcal{A} = (Q, Q_0, V, T)$ and $\mathcal{A}' = (Q', Q'_0, V', T')$ be two automata, and $\mathcal{A} \circ \mathcal{A}'$ be the synchronised product of these automata.*
Then the algorithm terminates and the constructed graph $OutAut = (\mathcal{C}, \mathcal{T})$ is such as :

(i) it is connected,
(ii) each element of $\mathcal{C}$ is a configuration of $\mathcal{A} \circ \mathcal{A}'$,
(iii) if a transition $C \rightarrow D$ is in $\mathcal{T}$, then it exists a path between the configurations C and D in $\mathcal{A} \circ \mathcal{A}'$,
(iiii) each non constructed state of $\mathcal{A} \circ \mathcal{A}'$ is covered by a constructed one.

Lemma 1 *Let $\mathcal{A} = (Q, Q_0, V, T)$ and $\mathcal{A}' = (Q', Q'_0, V', T')$ be two automata. Let $C = (w, w')$ be a configuration of $\mathcal{A} \circ \mathcal{A}'$ and $X \in$ CoMaxSync(C). We have: $S =$ Concat(C, X) is a configuration.*

PROOF:
Let $C = (w, w')$, $X \in$ *CoMaxSync(C)* and $S =$ *Concat(C, X)*.

1. $X = e$ **and** e **is an independent action that is enabled in the first automaton:**
   $S = (w.e, w')$ and C is a configuration, thus it is synchronised over the common actions. As we only add an independent action to C in order to construct S, then S is also synchronised over the common actions and thus S is a configuration.

2. $X = e$ **and** e **is an independent action that is enabled in the second automaton:**
   It is true for the same reason.

3. $X = (e_1, e_2)$ **and** (e_1, e_2) **are independent actions that are enabled in the first and the second automaton respectively:**
   It is true in the same way, with $S = (w.e_1, w'.e_2)$

4. $X = e$ **and** e **is an common action that is enabled for both automata:**
   $S = (w.e, w'.e)$ and C is a configuration, thus it is synchronised over the common actions. Here, we add a common action to C in order to construct S. But this action is added to the two words of C, thus S is synchronised over the common actions, and thus S is a configuration.

Lemma 2 *The algorithm terminates.*

PROOF:

The procedure *ConsProdSync* terminates because, for all configuration added to the product $OutAut$, there is only one call to *ConsProdSync*. Moreover, we add a configuration in $OutAut$, only if its associated state is not represented by an other configuration in $OutAut$, this implies that there is a finite number of configuration added to $OutAut$.

The procedure *Traverse* traverses of the sub-graph of $OutAut$. This graph may contains some cycles, but each time we add a transition which reached a already constructed state, we note that this transition is in a cycle. Thus, each cycle in $outAut$ contains at least one cyclic transition. Moreover, this procedure remember

all cyclic transitions that have been fired during the traversed and it never fire two times a same cyclic transition. Thus this procedure terminates.

The procedure *ConsHope* is not recursive and terminates.

Lemma 3 *Let $\mathcal{A} = (Q, Q_0, V, T)$ and $\mathcal{A}' = (Q', Q'_0, V', T')$ be two automata, and $\mathcal{A} \circ \mathcal{A}'$ be the synchronised product of these automata.*
The constructed graph $OutAut(\mathcal{C}, \mathcal{T})$ is such as each element of $\mathcal{C}$ is a configuration of $\mathcal{A} \circ \mathcal{A}'$.

PROOF:
There is two way to construct a configuration:
1. by concatenation with the preceding one (line A),
2. by cut (line C)

We make an induction proof; we suppose that all couples $(w, w') = C$ added to $OutAut$ until the n^{th} adding in $OutAut$ are configurations, and we prove that the couple $(v, v') = S$ added to $OutAut$ during the $n + 1^{th}$ adding in $OutAut$ is also a configuration.

Initialisation
The initial called is *ConsProdSync($(\varepsilon, \varepsilon), \mathcal{A}, \mathcal{A}'$).* Let S be the first element added to $OutAut$.

1. If S is constructed with the line A, then S = *Concat($(\varepsilon, \varepsilon), X$),* with $(\varepsilon, \varepsilon)$ is a configuration of $\mathcal{A} \circ \mathcal{A}'$ and X is in *CoMaxSync($(\varepsilon, \varepsilon)$).* According to lemma 1, S is a configuration of $\mathcal{A} \circ \mathcal{A}'$.

2. If S is constructed with the line B, then, according to the beginning of the procedure *ConsHope*, it exists a hope which has been transmitted to the configuration $(\varepsilon, \varepsilon)$, this is impossible.
   Thus S could not be constructed with the line B.

Recurrence
Let $S = (v, v')$ be the $n + 1^{th}$ element added to $OutAut$.

1. If S is constructed with the line A, then it is during the execution of a called of *ConsProdSync($C, \mathcal{A}, \mathcal{A}'$).* We notice that we only called *ConsProdSync* with element that has already been added in $OutAut$ (line *callProd1* and *callProd2*), thus according to the recurrence hypothesis, C is a configuration of $\mathcal{A} \circ \mathcal{A}'$. Moreover, we have S = *Concat(C, X),* with X is in *CoMaxSync(C).* Thus, according to lemma 1, S is a configuration of $\mathcal{A} \circ \mathcal{A}'$.

2. If S is constructed with the line B, then it is during the execution of a called of *ConsHope($C, H, OldHope, \mathcal{A}, \mathcal{A}'$).* We notice that we only called *ConsHope* with element that has already been added in $OutAut$ (line *callHope1*

and *callHope2*), thus according to the recurrence hypothesis, C is a configuration of $\mathcal{A} \circ \mathcal{A}'$. Moreover, we have S = *Concat(Cut(*C, n, m'*),e)* (if S = *Concat(Cut(*C, n', m*),e)*, it is the same reasoning), with:

- C is a configuration of $\mathcal{A} \circ \mathcal{A}'$,
- it exists $H = (e, \mathcal{A}, n, m) \in$ *CurrentHope(*C*)* (according to lines *callHope1* and *callHope2*),
- it exists $H' = (e, \mathcal{A}', n', m') \in OldHope$, i.e. in the set of transmitted hope of C (according to line *compa*)

We obtain :

- $C = (w, w')$, with $|w| = n$ and $|w'| = m$.
- it exists a configuration $C' = (u, u') \in \mathcal{A} \circ \mathcal{A}'$, such as $H' \in$ *CurrentHope(*C'*)* and $|u| = n'$, $|u'| = m'$ and C is a successor of C' (because H' has been transmitted to C) and thus $C' \subset C$.

Thus $S =$ *Concat(Cut(*C, n', m*),e)* = $(w.e, u'.e)$. Now, we must prove that S is a configuration of $\mathcal{A} \circ \mathcal{A}'$. Let us prove that the two words in S are in the languages of $\mathcal{A}$ and $\mathcal{A}'$ respectively:

- C is a configuration, thus w is in the language of $\mathcal{A}$. Moreover, $H = (e, \mathcal{A}, n, m) \in$ *CurrentHope(*C*)*, thus $w.e$ is also in the language of $\mathcal{A}$.
- C' is a configuration, thus u' is in the language of $\mathcal{A}'$. Moreover, $H' = (e, \mathcal{A}', n', m') \in$ *CurrentHope(*C'*)*, thus $u'.e$ is also in the language of $\mathcal{A}'$.

Now, let us prove that S is synchronised on common transitions:

> we have $C = (w, w')$, $C' = (u, u')$ and $C' \subset C$. Thus, it exists a couple non empty (x, x') such as $C' = (u.x, u'.x')$. So, $S = (w.e, u'.e) = (u.x.e, u'.e)$. The hopes of C' has been transmitted to C, thus only independent transitions have been fired between C' and C, thus x is a sequence of independent transitions, and thus as $C' = (u, u')$ is synchronised on common transition, then $(u.x, u')$ also is and then $S = (u.x.e, u'.e)$ also is.

Lemma 4 *Let $\mathcal{A} = (Q, Q_0, V, T)$ and $\mathcal{A}' = (Q', Q'_0, V', T')$ be two automata, and $\mathcal{A} \circ \mathcal{A}'$ be the synchronised product of these automata.*
Each full configuration C :
1. is in the constructed graph $OutAut(\mathcal{C}, \mathcal{T})$,
2. or it exists a full configuration S in $OutAut$ such as State(S) = State(C)

PROOF:
We make an induction proof on the level of the configurations.

A configuration is only added in $OutAut$ by lines A' or B'. Moreover, if a configuration is added to $OutAut$, then we call *ConsProdSync* with this configuration.

Moreover, the only way to subsitute a configuration by an other is in line *subst*.

Initialisation

The only full configuration with 0 level is $(\varepsilon, \varepsilon)$, and it added to $OutAut$ in the line 1 of the procedure *Main*. Later, if this configuration is substituted by another one, let us call it S, then this is done only if *State(S) = State(*$(\varepsilon, \varepsilon)$*)*

Recurrence

Let us prove that all configurations with level $L+1$ verifie the lemma. Let $S = (w, w') = (u_1 A_1 u_2 A_2 \dots u_n A_n u_{n+1}, u'_1 A_1 u'_2 A_2 \dots u'_n A_n u'_{n+1})$, where A_k are common transitions and u_k, u'_k are independent transitions, be a full configuration with its level equal to $L+1$. According to the definition of a full configuration, we have:

$|u_{n+1}| = |u'_{n+1}|$,

or ($|u_{n+1}| < |u'_{n+1}|$) and starting from S, the automaton $\mathcal{A}$ does not allow to fire any independent action,

or ($|u_{n+1}| > |u'_{n+1}|$) and starting from S, the automaton $\mathcal{A}'$ does not allow to fire any independent action.

1. $|u_{n+1}| = |u'_{n+1}| \neq 0$:
   $u_{n+1} = x.e$ and $u'_{n+1} = x.e'$, with e and e' independent actions of $\mathcal{A}$ and $\mathcal{A}'$ respectively.
   Thus it exists a configuration
   $C = (u_1 A_1 u_2 A_2 \dots u_n A_n x, u'_1 A_1 u'_2 A_2 \dots u'_n A_n x')$ which is full, because $|x| = |x'|$, and with $S =$ *Concat(*$C, (e, e')$*)*. C is full and its level is L, so by induction hypothesis: C has been added to $OutAut$ and thus it exists a call *ConsProdSync(*$D, \mathcal{A}, \mathcal{A}'$*)*, with D is a configuration such as *State(D) = State(C)*. We have $\{e, e'\} \in$ *CoMaxSync(D)*, thus S is added to $OutAut$ in line A' during this call. Later, if this configuration is substituted by an other, let us call it S', then this is done only if *State(S) = State(*S'*)*.

2. ($|u_{n+1}| < |u'_{n+1}|$) and starting from C, the automaton $\mathcal{A}$ does not allow to fire any independent action: $u'_{n+1} = x.e'$, with e' is an independent action of $\mathcal{A}'$:
   Thus it exists a configuration
   $C = (u_1 A_1 u_2 A_2 \dots u_n A_n u_{n+1}, u'_1 A_1 u'_2 A_2 \dots u'_n A_n x')$ which is full, because $|u_{n+1}| \leq |x'|$ and starting from C, the automaton $\mathcal{A}$ does not allow to fire any independent action. Moreover, we have
   $S =$*Concat(*C, e'*)*. C is full and its level is L, so by induction hypothesis, C has been added to $OutAut$, and thus it exists a call *ConsProdSync(*$D, \mathcal{A}, \mathcal{A}'$*)*, with D is a configuration such as *State(D) = State(*C*)*. We have $\{e\} \in$

CoMaxSync(C), thus S is added to $OutAut$ in line A' during this call. Later, if this configuration is substituted by an other, let us call it S', then this is done only if *State(S) = State(S')*.

3. ($|u_{n+1}| > |u'_{n+1}|$) and starting from S, the automaton $\mathcal{A}'$ does not allow to fire any independent action:
   We do the same reasoning as in the previously case.

4. $|u_{n+1}| = |u'_{n+1}| = 0$:
   We have $S = (w, w') = (u_1A_1u_2A_2 \dots u_nA_n, u'_1A_1u'_2A_2 \dots u'_nA_n)$. It exists a configuration $C = (u_1A_1u_2A_2 \dots u_n, u'_1A_1u'_2A_2 \dots u'_n)$, with $S =$ *Concat(C, A_n)*.

   (a) if C is full, then by induction hypothesis, C has been added to $OutAut$, and thus it exists a call *ConsProdSync($C, \mathcal{A}, \mathcal{A}'$)*. We have $\{A_n\} \in$ *CoMaxSync(C)*, thus S is added to $OutAut$ in line A' during this call. Later, if this configuration is substituted by an other, let us call it S', then this is done only if *State(S) = State(S')*.

   (b) If C is not full, then:
   - ($|u_n| < |u'_n|$) and starting from C, there is at least one independent transition that is enabled for the automaton $\mathcal{A}$, or
   - ($|u_n| > |u'_n|$) and starting from C, there is at least one independent transition that is enabled for the automaton $\mathcal{A}'$.
   Let us study the first case (the second case is similar):

   if ($|u_n| < |u'_n|$) and starting from C, there is at least one independent transition that is enabled for the automaton $\mathcal{A}$ (C is not full):
   We construct two full configurations: C_{min} and C_{max} such as $C_{min} \subset C \subset C_{max}$:

   If $|u_n| = 0$, then
   $C_{min} \leftarrow (u_1A_1u_2A_2 \dots A_{n-1}, u'_1A_1u'_2A_2 \dots A_{n-1})$
   Else
   $C_{min} \leftarrow (u_1A_1u_2A_2 \dots A_{n-1}u_n,$
   $\quad u'_1A_1u'_2A_2 \dots A_{n-1}f_1 \dots f_j)$ such as
   $u'_1A_1u'_2A_2 \dots A_{n-1}f_1 \dots f_j$ is in the language of $\mathcal{A}'$ and
   $f_1 \dots f_j.x = u'_n$ and $|u_n| = j$.
   C_{min} is full and its level is $< L$.

   $C_{max} \leftarrow (u_1A_1u_2A_2 \dots u_nx, u'_1A_1u'_2A_2 \dots u'_n)$, such as x is a sequence of independent transitions, $u_1A_1u_2A_2 \dots u_nx$ is in the language of $\mathcal{A}$ and: $|u_nx| = |u'_n|$ or $|u_nx| < |u'_n|$ but starting from C_{max}, the automaton $\mathcal{A}$ does not allow to fire any independent action

C_{max} is full and its level is L.

If $C_{min} = (u_{min}, u'_{min})$ then we have
$(A_n, \mathcal{A}, |u_{min}|, |u'_{min}|) \in$ *CurrentHope(*C_{min}*).*
Moreover, if $C_{max} = (u_{max}, u'_{max})$ then we have
$(A_n, \mathcal{A}', |u_{max}|, |u'_{max}|) \in$ *CurrentHope(*C_{max}*).*

By construction, it exists a path composed only by independent transitions between C_{min} and C_{max} in $\mathcal{A} \circ \mathcal{A}'$.
Thus the hopes of C_{min} has been transmitted to C_{max}, by successive calls of *ConsProdSync* or *Traverse*, and thus the configuration *Concat(Cut(*$C_{max}, |u_{min}|, |u'_{max}|$*), A*$_n$*)* = S will be constructed in the line B, then according to the algorithm, if it doesn't exist in $OutAut$, any configuration D such as *State(*D = *State(*S*),* the S is added to $OutAut$.

Lemma 5 *Let $\mathcal{A} = (Q, Q_0, V, T)$ and $\mathcal{A}' = (Q', Q'_0, V', T')$ be two automata, and $\mathcal{A} \circ \mathcal{A}'$ be the synchronised product of these automata.*
If it exists a transition $C \rightarrow D$ in $OutAut$, then it exists a path between C and D in $\mathcal{A} \circ \mathcal{A}'$.

PROOF:

Case 1: the transition is added in the line A'
This transition is: $C \xrightarrow{X}$ *Concat(*C, X*)*

1. If X is composed by only one action, independent or common, then the same transition exists in $\mathcal{A} \circ \mathcal{A}'$.

2. If X is composed by several actions, then $X = (e_1, e_2)$ such as starting from C, e_1 is enabled in $\mathcal{A}$ and e_2 is enabled in $\mathcal{A}'$. Thus it exists the path *State(*C*)* $\xrightarrow{e_1}$ *State(*C'*)* $\xrightarrow{e_2}$ *Concat(*C, X*)* in $\mathcal{A} \circ \mathcal{A}'$.

Case 2: the transition is added in the line B'
This transition is: $C_{init} \xrightarrow{X}$ *Concat(*C_{tmp}, X*).* It is added during the call *ConsHope(*$C, H, OldHope, \mathcal{A}, \mathcal{A}'$*),* with :
- C is a configuration
- $H = (e, aut, n, m) \in$ *CurrentHope(*C*)*
- $OldHope =$ *Hope(*C*)*

And it exists a hope $H' = (e, aut', n', m') \in$ *Hope(*C*),* because this transition is added.
Let us take the case in witch $aut = \mathcal{A}$ and $aut' = \mathcal{A}'$ (the reasoning is similar in the other case):

- We have $C_{init} =$ *Cut(*C, n', m'*)* = (u_{init}, u'_{init})
  According to the definitions of a hope and of a cut, C_{init} is a configuration

and $H' \in$ *CurrentHope(*C_{init}*)*
Moreover, as H' has been transmitted to C, then it exists a path composed only by independent transitions between C_{init} and C. Thus $C_{init} \subset C$ and $C = (u_{init}.v, u'_{init}.v')$

- Then, we have $C_{tmp} =$ *Cut(*C, n, m'*)* = (u_{init}, u'_{init}) = $(u_{init}.v, u'_{init})$
- C is a configuration, thus $u_{init}.v$ is in the language of $\mathcal{A}$, so it exists a path between C_{init} and C_{tmp} in $\mathcal{A} \circ \mathcal{A}'$, and thus it also exists a path between C_{init} and *Concat(*C_{tmp}, e*)*

Lemma 6 *$OutAut$ is connective*

PROOF
Let us make an induction proof over the number of constructed configuration.

Initialisation
The first constructed configuration is $(\varepsilon, \varepsilon)$. All graph with only one node is connective.

Recurrence
Let us suppose that the n first constructed configurations have their associated state that are organised in a connective graph.
We only add configuration in lines A'_t or A''_t or B'_t or B''_t, these configurations being constructed in lines A and B respectively.
Let S be a configuration which is the $n + 1^{th}$ constructed configuration.

1. If S is constructed with in line A, then in the lines A'_t or A''_t, the state associated to S is connected to the state of a configuration C that has been previously constructed. The result graph is connective.
2. If S is constructed with in line B, then in the lines B'_t or B''_t, the state associated to S is connected to the state of a configuration C_{init} that has been previously constructed. The result graph is connective.

PROOF OF THE THEOREM 1:
According to the lemmas 1, 2, 3, 4, 5 and 6, the algorithm is correct.

5 Test

For lack of time, we only implement a version with two automata of the previous algorithm, called 'Pikaia'. We don't test it on concrete examples. However, here a theorical example, because it has no semantic, resulting from this implementation. The figure 4 represents two input automata. We constructed with 'Pikaia' the whole

synchronised product of these automata, and the sub-graph of this product with our algorithm.

The result obtained is a reduction of 43 per 100 over the constructed states and of 69 per 100 over the constructed transitions.

```
partial construction : nbnodes=27  nbedges=26
complete construction : nbnodes=48  nbedges=85
edge reduction : 69.411765%
node reduction : 43.750000%
```

We have not implemented better version. It has been done in Marrella (Ambroise, Rozoy and Saquet 2003) for a reduction algorithm in another context; the reduction factor compared to tool as Spin is linear in the number of processors. We really think that the same factor is available with the present technique.

6 Conclusion

The algorithm that we have just presented and proved, constructs a sub-graph of the whole graph of synchronised product of the input automata and allows to detect all stable properties. This algorithm has been implemented and tested on several examples. Compared with other known algorithms that also reduce the number of constructed states and transitions, we abtain a gain about a factor two on the states and about a factor 3 on the transitions. This corresponds well to the intuitive idea of a reduction by assembled transitions, with a gain factor about the degre of the system parallelism.

Acknowledgment

The authors are indebted to anonymous referees that gave interested remarks.

References

ABDULLA, P. A. and ANNICHINI, A. and BENSALEM, S. and BOUAJJANI, A. and HABERMEHL, P. and LAKHNECH, Y., 1999, Verification of Infinite-State Systems by Combining Abstraction and Reachability Analysis, *Computer-Aided Verification, CAV 1999 (Nicolas Halbwachs and Doron Peled)*, **1633**, 146-159

AMBROISE, D., 2001, Marrella ou la simulation guidée d'alghorithmes répartis sur réseaux, *thèse de l'université de Paris 11*

AMBROISE, D. and ROZOY ,B., 1996, Marrella a tool to analyse the graph of states, *Parallel Processing Letters*, **6**, 583-594

AMBROISE, D. and ROZOY, B. and SAQUET, J., 2003, Deadlock Detection in Distributed Systems, *Proceedings of the ISCA 18th International Conference on Computers And Their Applications*, 210-213

AZEMA, P. and MICHEL, F. and VERNADAT, F., 1996, Covering Step Graph, *Lecture Notes in Computer Science*, **1091**, 516-?

BURCH, J.R. and CLARKE, E.M. and MCMILLAN, K.L. and DILL, D.L. and HWANG, L.J., 1992, Symbolic Model Checking: 10^{20} States and Beyond, *Information and Computation*, **98**, 142-170

CLARKE, E.M. and GRUMBERG, O. and PELED, D.A., 1999, Model Checking, *The MIT Press*

COOPER, R. and MARZULLO, K., 1991, Consistent Detection of Global Predicates, *Proceedings of 1991 ACM/ONR Workshop on Parallel and Distributed Debugging (ACM SIGPLAN Notices)*, 167-174

COUVREUR, J.-M., 1999, On-the-Fly Verification of Linear Temporal Logic, *Lecture Notes in Computer Science (Jeanette M. Wing and Jim Woodcock and Jim Davies)*, **1708**, 253-271

COUVREUR, J.-M. and POITRENAUD, D., 1999, Detection of Illegal Behaviours Based on Unfoldings, *Lecture Notes in Computer Science*, **1639**, 364-?

COUVREUR, J.-M. and POITRENAUD, D., 1996, Model Checking Based on Occurrence Net Graph, *IFIP Conference Proceedings (FORTE)*, **69**

DE SOUZA, M. L. and DE SIMONE, R., 1994, Partial (-Order) automata and Behavioural Equivalences, *INRIA Sophia Antipolis (France)*

DEVILLERS, R. and JANICKI, R. and KOUTNY, M. and LAUER, P.E., 1986, Concurrent and maximally concurrent evolution of nonsequential systems, *Theoretical Computer Science*, **43**, 213-238

ESPARZA, J., 1994, Model checking using net unfoldings, *Science of Computer Programming*, **23**, 151-195

ESPARZA, J. and ROMER, S., 1999, An Unfolding Algorithm for Synchronous Products of Transition Systems, *10th International Conference on Concurrency Theory, (LNCS, Springer-Verlag)*

ESPARZA, J. and HELJANKO, K., 2000, A New Unfolding Approach to LTL Model Checking, *Lecture Notes in Computer Science*, **1853**, 475-?

FERNANDEZ, J. C. and JARD, C. and JRON, T. and MOUNIER, L., 1992, On the fly verification of finite transition systems, *Formal Methods in System Design (Kluwer Academic Publishers)*, **1**, 251-273

GODEFROID, P., 1995, Partial-Order Methods for the Verification of Concurrent Systems, *Lecture Notes in Computer Science 1032 (Springer-Verlag)*

GODEFROID, P and WOLPER, P., 1991, Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties, *Proceedings of Computer Aided Verification (Springer-Verlag)*, **575**, 332-342

GODEFROID, P and WOLPER, P., 1991, A Partial Approach to Model Checking, *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, 406-415

HOLZMANN, G. J., 2002, Software Analysis and Model Checking, *Computer Aided Verification CAV 2002 (Copenhagen, Denmark)*

JARD, C. and JERON, T. and JOURDAN, G.V. and RAMPON, J.X., 1994, A general approach to trace-checking in distributed computing systems, 14^{th} *IEEE International Conference on DCS (Posnan)*, 396-403

LIU, Y.A. and STOLLER, S.D. and UNNIKRISHNAN, L., 2000, Efficient detection of global properties in distributed systems using partial-order methods, *Computer Aided Verification, CAV 2000*, **1855**

MICHEL, F. and VERNADAT, F., 1997, Covering Step Graph Preserving Failure Semantics, *18th Int. Conf on Application and Theory of Petri Nets*

PELED, D., 1994, Combining Partial Order Reductions with On-the-Fly Model-Checking, *Lecture Notes in Computer Science*, **818**, 377-?

SPIN *http://spinroot.com/spin/whatispin.html*

VALMARI, A.,1993, On-the-Fly Verification with Stubborn Sets, *Proc. 5th International Computer Aided Verification Conference*, 397-408

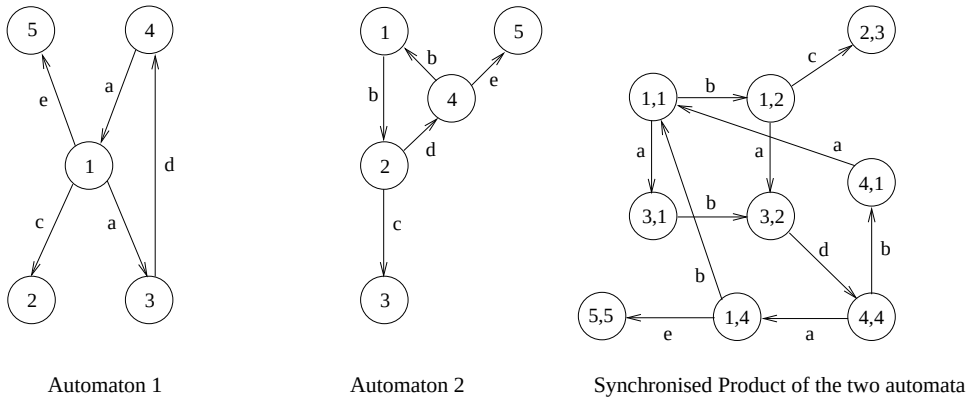

Figure 1: Synchronised Product of Automata

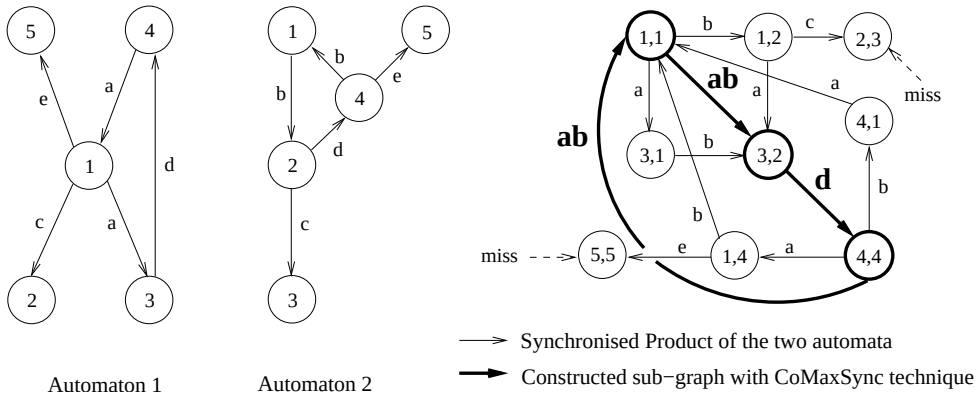

Figure 2: CoMaxSync technique

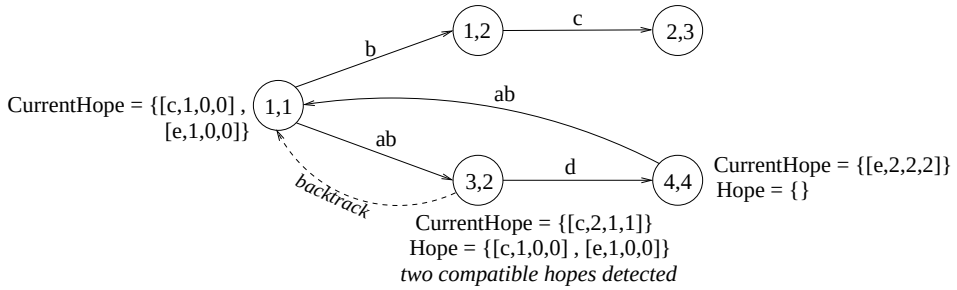

Figure 3: CoMaxSync with Hope backtrack

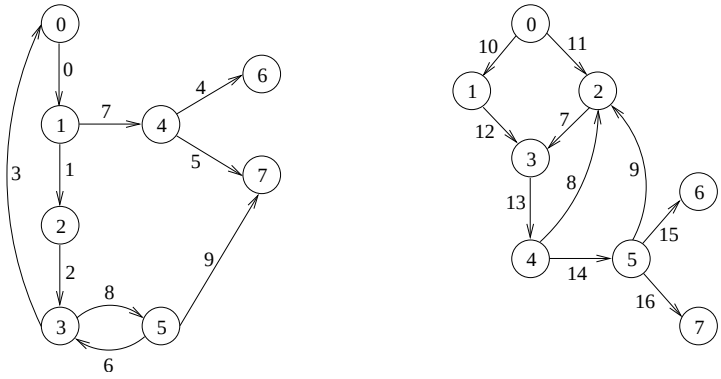

Figure 4: Test