Verification of Bakery algorithm variants for two processes

David Dedič [1], Robert Meolic [2]
[1]Nova Vizija d.o.o., Vrečerjeva ulica 8, SI-3310 Žalec
[2]Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, SI-2000 Maribor
david@vizija.si, meolic@uni-mb.si

***Abstract*—This paper is about Bakery algorithm for mutual exclusion. Three variants of the algorithm for two processes are discussed, formally modelled with a simple process algebra and then verified. Two methods of verification are given. In the first one, the processes representing behaviour of the algorithms are minimised with regard to testing equivalence and then examined. In the second approach, the properties are expressed with temporal logic ACTL and then verified using a model checker. Because we bounded the possible values of variables, the paper contibutes new results and knowledge to the well-known facts about Bakery algorithm.**

***Index Terms* – ACTL model checking, Bakery algorithm, Mutual exclusion, Testing equivalence**

I. Introduction

BAKERY algorithm for mutual exclusion is a simple and popular example of concurrent system discovered by L. Lamport in 1974 [1]. It does not depend upon any form of central control like semaphores. It implements mutual exclusion for N processes without relying on any lower-level mutual exclusion and works correct even if read and write operations on the same word of memory overlap. The algorithm for two processes can be easily derived from a more general algorithm for N processes. However, some authors have intentionally or inattentively simplified the original algorithm. It turns out that these modification have significant impact on the properties of the algorithm.

In Section 2 we describe Lamport's Bakery algorithm for N processes. In Section 3, three variants of Bakery algorithm for two processes are given and discussed. In Section 4, these variants are modelled with a simple process algebra. The processes representing behaviour of the algorithms are minimised with regard to testing equivalence and then examined. In Section 5, we present formal verification using ACTL model checking. The paper concludes with some remarks.

II. Description of Bakery Algorithm

Lamport's Bakery algorithm for mutual exclusion given in Fig. 1 is designed for a system with N processes. Each of them includes a critical section which must not be executed concurrently with the critical sections of the other processes. A shared memory is used for access arbitration. The arbitration follows the principle of serving ***customers*** at a ***bakery***, where the processes are the customers which can be served only one at a time. Each process receives a ***ticket number*** as the costumers in a store would, and the process with the lowest number is allowed to enter the critical section. This is an infinite-state system since ticket numbers may grow arbitrarily large. When a process leaves its critical section, it gets ticket number 0, which indicates that the process is not interested in accessing the critical section.

```
PROCESS Pi
loop forever
i0: <non-critical section>
i1: Choosing(i) := 1;
i2: Number(i) := 1 + max(Number[1],…,Number[N]);
i3: Choosing(i) := 0;
    for j in 1..N do begin
i4:     loop
            exit when Choosing(j) == 0;
            end loop;
i5:     loop
con:        exit when Number(j) == 0 or Number(i) < Number(j) or
                      (Number(i) == Number(j) and i < j);
        end loop;
    end;
i6: <critical section>
i7: Number(i) := 0;
end loop;
```

Fig. 1: Bakery algorithm for one out of N processes

By formal verification of Bakery algorithm, the following properties can be shown:

- ***Mutual exclusion***: at any time, only one process can be in its critical section.
- ***Accessibility:*** if a process expresses interest in entering the critical section, it will eventually do so.
- ***One-bounded overtaking:*** if process Pi intends to enter the critical section, any other process can enter the critical section at most once before Pi does.

The Bakery algorithm was the first solution that satisfied all three properties above for the general case of N processes.

III. Bakery Algorithm Variants for Two Processes

Original variant of Bakery algorithm. From the general Bakery algorithm the algorithms for the case of two processes can be derived. Processes P1-BAKERY and P2-BAKERY use variables *c1*, *c2*, *n1*, and *n2* instead of arrays *Choosing* and *Number*. The algorithms for both processes are shown in Fig. 2 and they differ only in conditions *con1* and *con2*.

The working principle is the same as in the general Bakery algorithm. Suppose that both processes are in the non-critical section, *n1*=*n2*=0, and process P1-BAKERY wants to enter its critical section. It sets variable *c1* to 1 and performs addition in line *a2*. It gets the ticket number with value 1. Afterwards, process P1-BAKERY enters its critical section as it has the lowest ticket number and condition *con1* is true. However, process P2-BAKERY also wants to enter the critical section. The value *n2* is set to 2, because value of *n1* is 1. Thus process P2-BAKERY is being denied entering critical section because the condition *con2* is false. It must wait in loop *b4*. After process P1-BAKERY leaves the critical section, it sets *n1*=0. Process P2-BAKERY has now the lowest ticket number and can enter its critical section.

```
PROCES P1-BAKERY                      PROCES P2-BAKERY
loop forever                          loop forever
a0: <non-critical section>            b0: <non-critical section>
a1: c1 := 1;                          b1: c2 := 1;
a2: n1 := n2 + 1;                     b2: n2 := n1 + 1;
a3: c1 := 0;                          b3: c2 := 0;
a4: loop                              b4: loop
      exit  when c2==0;                     exit  when c1==0;
    end loop;                             end loop;
a5: loop                              b5: loop
con1: exit  when n2==0 or n1<=n2;     con2: exit  when n1==0 or n2<n1;
    end loop;                             end loop;
a6: <critical section>                b6: <critical section>
a7: n1 := 0;                          b7: n2 := 0;
end loop;                             end loop;
```

Fig. 2: Bakery algorithm for two processes

What happens if processes P1-BAKERY and P2-BAKERY want to enter their critical sections simultaneously? If both processes operate with the same speed, steps *a2* and *b2* are calculated simultaneously resulting in value 1 for both variables *n1* and *n2*. Both processes have the lowest (equal) ticket number. Due to different conditions for exit in loops *a5* and *b5*, only the first process will enter its critical section. If conditions *con1* and *con2* were the same, the system would either deny or allow both processes to enter its critical section.

A simplified Bakery algorithm. In [2], M. Ben-Ari gives the variant of Bakery algorithm shown in Fig. 3. Variables *c1* and *c2*, which are important for assuring mutual exclusion, are removed and their role is compensated by variables *n1* and *n2*, respectively. Moreover, assignments *a3* and *b3* and loops *a4* and *b4* are removed. We will denote the processes implementing this variant of the algorithm by P1-BEN-ARI and P2-BEN-ARI.

An even more simplified Bakery algorithm. In [3], authors present an even more simplified variant of Bakery algorithm, where variables *c1* and *c2* are simply missing and not substituted by any mechanism. Processes implementing this variant of the algorithm, which do not contain lines *a1* and *b1*, but are otherwise the same as shown in Fig. 3, will be denoted by P1-STEP and P2-STEP.

```
PROCES P1-BEN-ARI                     PROCES P2-BEN-ARI
loop forever                          loop forever
a0: <non-critical section>            b0: <non-critical section>
a1: n1 := 1;                          b1: n2 := 1:
a2: n1 := n2 + 1;                     b2: n2 := n1 + 1;
a5: loop                              b5: loop
con1: exit  when n2==0 or n1<=n2;     con2: exit  when n1==0 or n2<n1;
    end loop;                             end loop;
a6: <critical section>                b6: <critical section>
a7: n1 := 0;                          b7: n2 := 0;
end loop;                             end loop;
```

Fig. 3: Simplified Bakery algorithm for two processes from [2]

IV. Representing algorithms with processes

Process algebrae are widely used formalisms for modelling and verification of concurrent systems. Systems are described as a set of communicating ***processes***. We will use an algebraic approach which is similar to CCS and has some notation from CSP. Processes are labelled directed graphs. Graph nodes are called ***states***. An edge from state *p* to state *q* labelled with action α is called an ***α-transition*** or shortly a ***transition*** from state *p* to state *q*. If there exists an α-transition from a given state, we say that in this state the process can ***perform*** α-transition or that it can **perform** action α. The set of all actions which a process can perform is called the ***alphabet*** of the process. A sequence of transitions in the process is called a ***path***. A state without outgoing transitions is a ***deadlocked state***. The alphabet of a process includes a ***silent action τ***, which is used to model ***internal transitions***. All actions others than τ are called ***visible*** actions and are divided into input and output actions. The name of an ***output action*** terminates by '!', e.g. *a!*, *b!*,... The name of an ***input action*** terminates by '?', e.g. *a?*, *b?*,... The pairs of actions *a?* and *a!*, *b?* and *b!* etc. are called ***complementary actions***.

Processes can run concurrently forming a compound system. It is assumed that processes perform their transitions asynchronously. However, each process can perform a visible action only if one of the other processes in the system concurrently performs the complementary action. The only exceptions of this rule are special ***external actions*** used to interact with the system environment, which can be performed by a process independently of the other processes.

An important concept in process algebrae is testing equivalence [4]. An ***observer*** *O* is a process containing special external action *w!*, which denotes success of the observation. A concurrent run of process *P* and observer *O* is called an ***observation***, and it is ***successful*** if the action *w!* is performed. Process *P* ***must satisfy*** the observer *O* if the observation is always successful. Process *P* ***may satisfy*** the observer *O* if the observation may be successful. Two processes are ***must equivalent*** iff they must satisfy the same sets of observers. They are ***may equivalent*** iff they may satisfy the same sets of observers. Must equivalence does not imply may equivalence although for a particular observer, if a process must satisfy the observer, then also may satisfy it. Finally, the processes are ***testing equivalent*** iff they are must and may equivalent.

Processes representing variables. Variables *n1*, *n2*, *c1*, and *c2* are represented with processes composed of states, such that each state represents one possible value of the variable. Variables *c1* and *c2* can only have value 0 or 1, whereas variables *n1* and *n2* may have value 0, 1, or 2. In Fig. 4, process N1 is given in textual form, which is used on a computer. There, an α-transition from state *p* to *q* is denoted by **p = α.q**. Processes representing other variables are similar.

```
N10 = n1r0!.N10 + n1w0?.N10 + n1w1?.N11 + n1w2?.N12
N11 = n1r1!.N11 + n1w0?.N10 + n1w1?.N11 + n1w2?.N12
N12 = n1r2!.N12 + n1w0?.N10 + n1w1?.N11 + n1w2?.N12
```

Fig. 4: Process N1 representing variable *n1*

Processes for addition. Addition is implemented as a conditional value setting. We used two different solutions. In the first case, process NPLUS performs both additions. In the second solution, there are two separate processes, N1PLUS and N2PLUS**,** for the calculation of *n1* and *n2*, respectively. The main difference between these solutions is that in the last case concurrent operation on both variables is possible. Processes for addition are shown in Fig. 5 and 6.

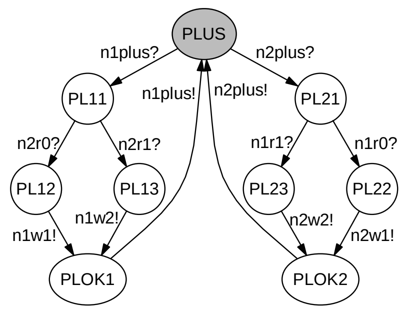

Fig. 5: Process NPLUS

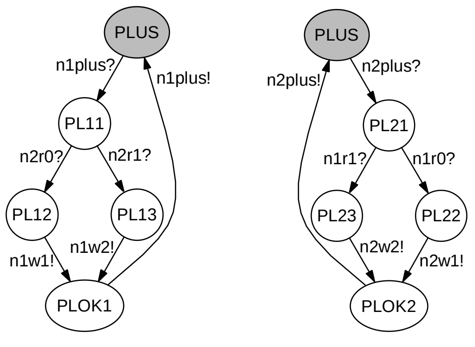

Fig. 6: Processes N1PLUS and N2PLUS

***Processes* P1-BAKERY *and* P2-BAKERY.** These processes represent the Bakery algorithm. They are shown in Fig. 7. The processes are different because of different conditions *con1* and *con2*. In the model, conditions are optimised as the values for *n1* and *n2* are bounded. Actions *n1plus!* and *n2plus!* launch the addition and it is not determined, if the implementation with one or two processes will be used.

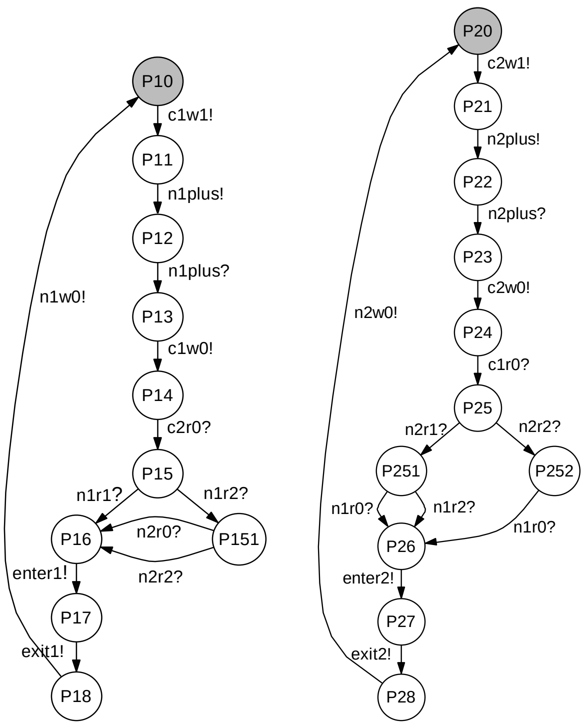

Fig. 7: Processes P1-BAKERY and P2-BAKERY

***Processes* P1-BEN-ARI, P2-BEN-ARI, P1-STEP, *and* P2-STEP.** These processes are similar to those representing original Bakery algorithm. Processes P1-BEN-ARI and P2-BEN-ARI are presented in Fig. 8. Processes P1-STEP and P2-STEP are almost the same, but their initial states are P11 and P21, respectively, and they also return into these states.

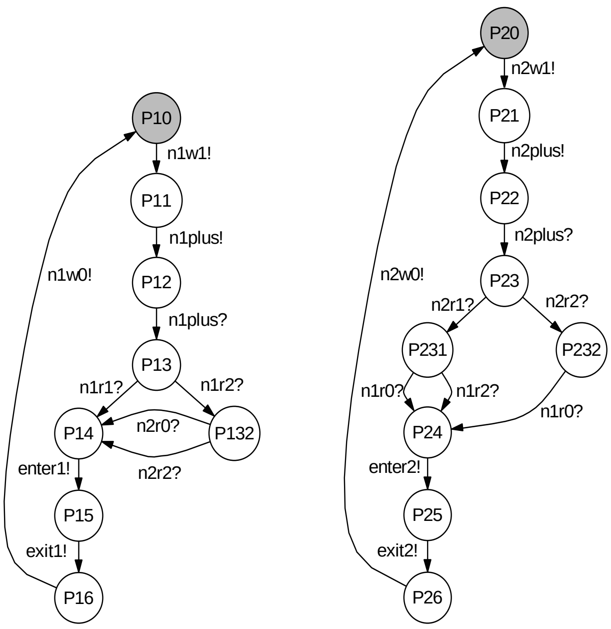

Fig. 8: Processes P1-BEN-ARI and P2-BEN-ARI

To obtain the behaviour of each variant of Bakery algorithm, separate components are composed in parallel, and the behaviour of the whole system is then represented with a single process. In this way, we create processes BAKERY, BEN-ARI, STEP, which represent the systems using process NPLUS, and processes BAKERY2, BEN-ARI2, and STEP2, which represent the systems where processes N1PLUS and N2PLUS are used. The obtained processes are large and are not directly usable for reasoning about the algorithms (see Table 1). One solution to this problem is to create smaller, but testing equivalent processes. They contain all externally observable properties and are shown in Fig. 9, 10, 11, and 12. It turns out that processes BAKERY and BAKERY2 are testing equivalent and therefore process TEST-BAKERY adequately represents both systems. The same situation is with processes BEN-ARI and BEN-ARI2.

TABLE 1: THE SIZE OF PROCESSES REPRESENTING ALGORITHMS

Process	States	Transitions
BAKERY	218	381 (52 visible +329 internal)
BAKERY2	292	521 (60 visible +461 internal)
BEN-ARI	148	256 (44 visible +212 internal)
BEN-ARI2	214	383 (52 visible +331 internal)
STEP	112	191 (36 visible +155 internal)
STEP2	242	451 (108 visible +343 internal)

From the obtained processes, interesting conclusions can be carried out. The algorithms behave differently. All the systems except STEP2 are may equivalent but they are not must equivalent. For example, an observer able to perform sequences *enter1?*, *w!* and *enter2?*, *w!* must satisfy processes BAKERY and STEP, but not process BEN-ARI. By analysing the processes, we can also try to answer the questions about mutual exclusion, accessibility, and one-bounded overtaking.

Mutual exclusion: We can easily conclude that all the systems except STEP2 assure mutual exclusion.

Accessibility: Only STEP and STEP2 assure accessibility, because the other systems contain deadlocked state. This happens because variables *n1* and *n2* are bounded in our model.

One-bounded overtaking: The validity of this property cannot be established by analysing the minimised processes.

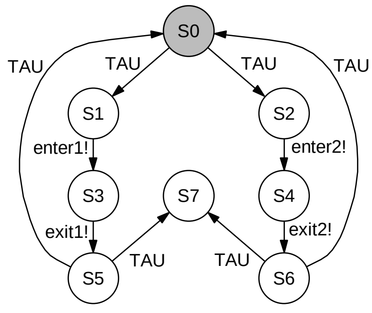

Fig. 9: Process TEST-BAKERY

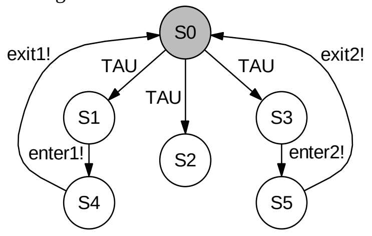

Fig. 10: Process TEST-BEN-ARI

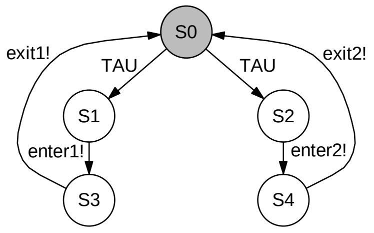

Fig. 11: Process TEST-STEP

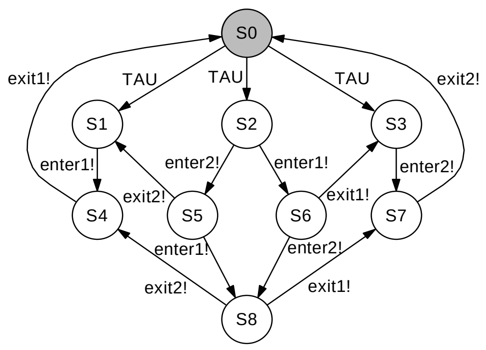

Fig. 12: Process TEST-STEP2

V. MODEL CHECKING WITH ACTL

We will specify circuit properties using ***action computation tree logic (ACTL)***, which is a propositional branching time temporal logic. We will use a variant of ACTL with *unless* operator [5].

ACTL formulae are composed of constants *true* and *false*, standard Boolean operators *NOT*, *AND*, *OR*, ***action formulae*** written in brackets, ***path quantifiers*** **E** ("there exists a path") and **A** ("for all paths"), and ***temporal operators*** **U** ("until"), **W** ("unless"), **X** ("for the next transition"), **F** ("for some transition in the future"), and **G** ("for all transitions in the future"). Action formulae may contain visible actions, silent action τ, and standard Boolean operators. ACTL formulae are evaluated in states. A state where ACTL formula φ is valid will be called φ-state. An action for which action formula χ is valid will be called χ-action. A transition from state *p* to state *q* where action formula χ is valid for the action performed during this transition and ACTL formula φ is valid in state *q* will be called (χ,φ)-transition. The meaning of temporal operators is given as follows:

- Formula $\mathbf{X}\{\chi\}\ \varphi$ is valid on path π iff the first transition on this path is a (χ,φ)-transition.
- Formula $\mathbf{F}\{\chi\}\ \varphi$ is valid on path π iff there exists a (χ,φ)-transition on this path.

- Formula **G** φ {χ}is valid on path π iff ACTL formula φ is valid in the first state of this path and all transitions on the path are (χ ,φ)-transitions.
- Formula [φ {χ}**U**{χ'} φ'] is valid on path π iff ACTL formula φ is valid in the first state of this path and the path begins with a finite sequence of (χ ,φ)-transitions followed by a (χ' ,φ')-transition.
- Formula [φ {χ}**W**{χ'} φ'] is valid on path π iff formula [φ {χ}**U**{χ'} φ'] is valid on this path or formula **G** φ {χ} is valid on this path.

The listed formulae are called path formulae. Each path formula must be immediately preceded by a path quantifier. Path quantifier **E** requires that the property expressed by the path formula is valid for at least one path starting in the given state. Path quantifier **A** requires that the property expressed by the path formula is valid for all paths starting in the given state. In a deadlocked state, formulae **EG** φ {χ}, **AG** φ {χ}, **E**[φ {χ}**W**{χ'} φ'], and **A**[φ {χ}**W**{χ'} φ'] are valid only if the state is a φ -state. Formulae **EX**{χ} φ, **AX**{χ} φ, **EF**{χ} φ, **AF**{χ} φ, **E**[φ {χ}**U**{χ'} φ'], and **A**[φ {χ}**U**{χ'} φ'] are invalid in all deadlocked states. We will take that an ACTL formula is valid in process *P* iff it is valid in its initial state.

In ACTL formulae, we will omit the constant *true* in some cases, for example: **AG** φ = **AG** φ {*true*} and **E**[{χ} **U** {χ'}] = **E**[*true* {χ}**U**{ χ'} *true*]. We also use commonly used abbreviation [χ] φ. ACTL formula [α] φ is valid in the given state iff all α-transitions from that state are (α ,φ)-transitions. To enable the verification of interesting properties, we introduced external actions *request1!* and *request2!* in processes P1 and P2, respectively. We added a transition with this action right after the addition completes. Then, we express mutual exclusion, accessibility, and one-bounded overtaking with the following ACTL formulae:

1. There is no deadlock state:
   AG AF {*true*}
2. If process P1 (P2) enters its critical section, then process P2 (P1) cannot enter its critical section until P1 exits:
   AG [*enter1!*] **A**[{*NOT enter2!*} **U** {*exit1!*}]
   AG [*enter2!*] **A**[{*NOT enter1!*} **U** {*exit2!*}]
3. If process P1 (P2) expresses interest in entering its critical section, it will eventually do so:
   AG [request1!] **AF** {enter1!}
   AG [request2!] **AF** {enter2!}
4. After process P1 (P2) intends to enter its critical section, process P2 (P1) can enter its critical section at most once before P1 (P2) does.
   AG [*request1!*] *NOT* **E**[{*NOT enter1!*} **U** {*enter2!*}
   E[{*NOT enter1!*} **U** {*enter2!*}]]
   AG [request2!] *NOT* **E**[{*NOT enter2!*} **U** {*enter1!*}
   E[{*NOT enter2!*} **U** {*enter1!*}]]

The results of model checking were in accordance with our expectations after the analysis of testing equivalent processes. Systems STEP and STEP2 do not have deadlocked states, while the others have one. All the systems except STEP2 assure mutual exclusion and only STEP and STEP2 assure accessibility. As a bonus, we also found out that all the systems have the property of one-bounded overtaking.

Model checking turns out to be very elegant, flexible and powerful verification method. An additional advantage are useful counterexamples. For example, the following two explain, how system BENARI can reach deadlocked state and why system STEP2 does not assure mutual exclusion.

AG AF {*true*} ==> **FALSE**

```
(P10<P1BENARI>,P20<P2BENARI>,N10<N1>,N20<N2>,PLUS<NPLUS>) → τ →
(P10<P1BENARI>,P21<P2BENARI>),N10<N1>,N21<N2>,PLUS<NPLUS>) → τ →
(P10<P1BENARI>),P22<P2BENARI>,N10<N1>,N21<N2>,PL21<NPLUS>) → τ →
(P11<P1BENARI>,P22<P2BENARI>,N11<N1>,N21<N2>,PL21<NPLUS>) → τ →
(P11<P1BENARI>,P22<P2BENARI>,N11<N1>,N21<N2>,PL23<NPLUS>) → τ →
(P11<P1BENARI>,P22<P2BENARI>,N11<N1>,N22<N2>,PLOK2<NPLUS>) → τ →
(P11<P1BENARI>,P220<P2BENARI>,N11<N1>,N22<N2>,PLUS<NPLUS>) → τ →
(P12<P1BENARI>,P220<P2BENARI>,N11<N1>,N22<N2>,PL11<NPLUS>) → request2!
→
(P12<P1BENARI>),P23<P2BENARI>,N11<N1>,N22<N2>,PL11<NPLUS>) → τ →
(P12<P1BENARI>,P232<P2BENARI>,N11<N1>,N22<N2>,PL11<NPLUS>) .
```

AG [*enter1!*] **A**[{*NOT enter2!*} **U** {*exit1!*}] ==> **FALSE**

```
(P11<P1STEP>,P21<P2STEP>,N10<N1>,N20<N2>,PLUS<N1PLUS>,PLUS<N2PLUS>) → τ →
(P11<P1STEP>,P22<P2STEP>,N10<N1>,N20<Y2>,PLUS<N1PLUS>,PL21<N2PLUS>) → τ →
(P11<P1STEP>,P22<P2STEP>,N10<N1>,N20<N2>,PLUS<N1PLUS>,PL22<N2PLUS>) → τ →
(P12<P1STEP>,P22<P2STEP>,N10<N1>,N20<N2>,PL11<N1PLUS>,PL22<N2PLUS>) → τ →
(P12<P1STEP>,P22<P2STEP>,N10<N1>,N20<N2>,PL12<N1PLUS>,PL22<N2PLUS>) → τ →
P12<P1STEP>,P22<P2STEP>,N10<N1>,N21<N2>,PL12<N1PLUS>,PLOK2<N2PLUS>) → τ →
P12<P1STEP>,P220<P2STEP>,N10<N1>,N21<N2>,PL12<N1PLUS>,PLUS<N2PLUS>) → request2! →
(P12<P1STEP>,P23<P2STEP>,N10<N1>,N21<N2>,PL12<N1PLUS>,PLUS<N2PLUS>) → τ →
(P12<P1STEP>,P231<P2STEP>,N10<N1>,N21<N2>,PL12<N1PLUS>,PLUS<N2PLUS> → τ →
(P12<P1STEP>,P24<P2STEP>,N10<N1>,N21<N2>,PL12<N1PLUS>,PLUS<N2PLUS>) → τ →
(P12<P1STEP>,P24<P2STEP>,N11<N1>,N21<N2>,PLOK1<N1PLUS>,PLUS<N2PLUS>) → τ →
(P120<P1STEP>,P24<P2STEP>,N11<N1>,N21<N2>,PLUS<N1PLUS>,PLUS<N2PLUS>) → request1! →
(P13<P1STEP>,P24<P2STEP>,N11<N1>,N21<N2>,PLUS<N1PLUS>,PLUS<N2PLUS>) → τ →
(P14<P1STEP>,P24<P2STEP>,N11<N1>,N21<N2>,PLUS<N1PLUS>,PLUS<N2PLUS>) → enter1! →
(P15<P1STEP>,P24<P2STEP>,N11<N1>,N21<N2>,PLUS<N1PLUS>,PLUS<N2PLUS>) → enter2! →
(P15<P1STEP>,P25<P2STEP>,N11<N1>,N21<N2>,PLUS<N1PLUS>,PLUS<N2PLUS>) .
```

VI. Conclusion

This paper shows, how the properties of mutual exclusion algorithms can be formally verified. The analysis of the compound process can answer all questions, but the problem is the size of the process. Minimisation can help, but it hides some properties and it is also a complex operation. For example, our tool cannot automatically find testing equivalent processes and therefore we needed about one hour of hard work for each of them. On the other hand, ACTL model checking is an effective and fully automatic method. There, the most complex task is the construction of ACTL formulae.

Another variant of Bakery algorithm is also known to us. It is obtained by assuming line *i2* in Fig. 1 to be atomic operation. Then, processes never have equal ticket numbers and therefore the condition for entering the critical section can be simplified. However, due to Lamport the result of such a rigorous modification is an emasculated algorithm and it lacks all beauty of the original.

References

[1] L. Lamport. *A new solution of Dijkstra's concurrent programming.* Communications of the ACM 17(8):453-455, August 1974.

[2] M. Ben-Ari. *Principles of concurrent and distributed programming.* Prentice Hall International (UK), 1990.

[3] N. S. Bjørner et al. *Verifying temporal properties of reactive systems: A STeP tutorial.* Formal Methods in System Design, 16(3):227-270, June 2000.

[4] R. de Nicola and M. C. B. Hennessy. *Testing equivalences for processes.* Theoretical Computer Science, 1-2(34):83-133, November 1984.

[5] R. Meolic, T. Kapus, and Z. Brezočnik. *An action computation tree logic with* unless *operator.* Submitted for publication..