University of Helsinki
Department of Computer Science
Series of Publication C, No. C-2000-3

# Verification of progress properties in a shared-memory concurrent system represented as a labeled transition system

Martti Tienari and Päivi Kuuppelomäki

Helsinki, February 2000

Report C-2000-3

University of Helsinki
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 Helsinki, FINLAND

**Verification of progress properties in a shared-memory concurrent system represented as a labeled transition system**

Martti Tienari and Päivi Kuuppelomäki

Department of Computer Science
P.O.Box 26, FIN-00014 University of Helsinki, Finland
Martti.Tienari@cs.Helsinki.FI

Series of Publications C, Report C-2000-3
Helsinki, February 2000, 26 pages

## Abstract

A shared memory concurrent system can be modeled as a labeled transition system. The system is assumed to be finite. Verification of progress properties of this kind of systems is studied.

We present methods to check "leads-to" type of progress properties from global state graphs for both weakly and strongly fair executions. We also study a new method, applicable for two-process systems under weak fairness, to check progress properties of a system from the corresponding bisimilar minimized system. A novel divergence preserving bisimilarity is needed to make this possible. Two simple mutual exclusion protocols are analyzed to demonstrate the workings of the latter method.

**Computing Reviews (1998) Categories and Subject Descriptors:**
F.3.1 Specifying and Verifying and Reasoning about Programs
D.2.4 Program Verification

**General terms:** Verification

**Additional Key Words and Phrases:** Equivalences, Fairness, Labeled Transition System, Liveness

# 1 Introduction

In an interesting case study [Walk89] several mutual exclusion algorithms were analyzed. The systems were first modelled with CCS, translated to lts, some properties were then expressed in modal logic and subsequently verified using a model checker available in the Concurrency Workbench [ClPS89]. In addition to a safety property "mutual exclusion" also a liveness property "if an entry to the critical section is requested it is always eventually granted" was studied in [Walk89]. There the lts's were minimized with respect to observation equivalence [Miln89], which does not preserve the divergences, in order to improve the efficiency of model checking. A nonstandard special interpretation of liveness properties was needed in order to be able to verify these using the Concurrency Workbench.

In her Ph.D. dissertation [Elor94] Jaana Eloranta studied, based on [Walk89], the same mutual exclusion algorithms. She employed Basic LOTOS in process modelling and, instead of using model checking, the systems were reduced or minimized with respect to some divergence preserving equivalences after suitable event label hidings. By visual inspection of the resulting reduced lts's — the number of states after reduction was between 3 and 43 and the number of transitions was between 7 and 85 — the same safety and liveness properties as in Walker's work were seen to hold. However, there were several divergences in the reduced systems which could, in principle, endanger the liveness properties if Walker's clever restricted interpretation of liveness would not have been employed. As a step towards a more general analysis the divergences were interpreted in [Elor94, pages 88–89] in terms of real system properties and it was argued that they cannot appear in reality.

In the current paper our goal is, based on the work in [Walk89] and [Elor94], to find semiautomatic methods to analyze liveness properties of shared memory concurrent systems. This requires introduction of fairness constraints in our analysis. We also want to study the following problem: Assuming we employ reduction with respect to a divergence preserving behavioural equivalence to analyze a concurrent system, how can we analyze adequately the liveness (progress) properties of the system?

Related work to our developments in Section 3 has been done by model checking researchers ([MaPn84], [LiPn84], [MaPn95]). We are also aware of two published approaches ([VaSe96], [Gian99]) to study liveness properties of systems by inspecting systems minimized with respect to some behavioural equivalences.

In the following, after introducing some notation and a novel version of a divergence preserving bisimilarity, we describe the general principles how to model shared-memory concurrent systems in Basic LOTOS process algebra and as labeled transition systems. The structure of these lts's is then analyzed as a graph of strongly connected components, and properties of fair execution paths are analyzed under both weak and strong fairness. Preservation of the progress properties is also studied after hiding appropriate labels and minimizing the lts with respect to a novel divergence preserving bisimilarity. It turns out that, in case of two concurrent processes executing under weak fairness assumption, we are able to deduce progress properties of the modeled system by looking at a minimized system. Two simple case studies demonstrate the technique: We analyze progress properties of Petersons's and Lamport's mutual exclusion algorithms for two processes.

# 2 Basic definitions

In this paper we are using the following definitions of basic concepts.

**Definition 1** A *labeled transition system* (lts) is a structure $K = (S, Act_\tau, T, s0)$ where $S$ is a finite set of *states*, $Act$ is a finite set of *visible actions*, $\tau$ is the *invisible action*, we denote $Act_\tau = Act \cup \{\tau\}$, $T \subseteq S \times Act_\tau \times S$ is a *transition relation*, and $s0$ is the *initial state*, $s0 \in S$. $\Box$

**Definition 2** Let $K = (S, Act_\tau, T, s0)$ be an lts and $a \in Act_\tau$, $t = a_1 \cdots a_n \in Act_\tau^*$, $n \geq 0$. Then

- $s \xrightarrow{a} s'$ iff $(s, a, s') \in T$,
- $s \xrightarrow{t} s'$ iff $\exists s_1, \cdots, s_{n+1} \in S, n \geq 0$: $s = s_1 \wedge s_1 \xrightarrow{a_1} s_2 \wedge \cdots \wedge s_n \xrightarrow{a_n} s_{n+1} \wedge s_{n+1} = s'$,
- $s \overset{t}{\Longrightarrow} s'$ iff $t \in Act^*$ and $s \xrightarrow{t'} s'$ holds with such an action sequence $t' \in Act_\tau^*$ that $t$ is obtained from $t'$ by removing all $\tau$-actions. When $t$ is an empty sequence $\varepsilon$, $\overset{t}{\Longrightarrow}$ is reflexive, i.e., $s \overset{\varepsilon}{\Longrightarrow} s$. $\Box$

**Definition 3** Let $K = (S, Act_\tau, T, s0)$ be an lts. A state $s \in S$ is *divergent* iff $div(s)$ where $div(s) =_{def} \{\exists$ an infinite sequence $\{s_1, s_2, \cdots\}$ such that $s = s_1 \wedge (\forall n \geq 1 : s_n \xrightarrow{\tau} s_{n+1})\}$.
State $s \in S$ has a *$\tau$-loop* iff $s \xrightarrow{\tau} s$ and an *$\varepsilon$-cycle* iff $\exists s' \in S$: $s' \neq s \wedge s \overset{\varepsilon}{\Longrightarrow} s' \wedge s' \overset{\varepsilon}{\Longrightarrow} s$. (A *$\tau$-cycle* is either a $\tau$-loop or a $\varepsilon$-cycle.)

A state $s \in S$ is *strictly divergent* iff $sdiv(s)$ where
$sdiv(s) = (s, \tau, s) \in T \vee (\exists s' \in S : s' \neq s \wedge s \stackrel{\varepsilon}{\Longrightarrow} s' \wedge s' \stackrel{\varepsilon}{\Longrightarrow} s)$,
i.e., iff the state $s$ has a $\tau$-loop or an $\varepsilon$-cycle. $\Box$

**Definition 4** Let $K_i = (S_i, Act_i \cup \{\tau\}, T_i, s0_i)$, $i = 1, 2$, be two lts's, $SD =_{def} \{(s_1, s_2) \in S_1 \times S_2 \mid sdiv(s_1) = sdiv(s_2)\}$ , and $Act = Act_1 \cup Act_2$. A relation $R \subseteq S_1 \times S_2$ is a $SD$-bisimulation (*strict divergence bisimulation*) iff $R \subseteq SD$ and for each $(s_1, s_2) \in R$ and for each $u \in Act \cup \{\varepsilon\}$
i) whenever $s_1 \stackrel{u}{\Longrightarrow} s_1'$, then for some $s_2' \in S_2$, $s_2 \stackrel{u}{\Longrightarrow} s_2'$ and $(s_1', s_2') \in R$, and
ii) whenever $s_2 \stackrel{u}{\Longrightarrow} s_2'$, then for some $s_1' \in S_1$, $s_1 \stackrel{u}{\Longrightarrow} s_1'$ and $(s_1', s_2') \in R$.

Lts's $K_1$ and $K_2$ are *$SD$-bisimilar*, written $K_1 \approx_{SD} K_2$ iff $(s0_1, s0_2) \in R$ for some $SD$-bisimulation $R$. $\Box$

**Lemma 5** [Elor94] If states $s_1$ and $s_2$ are in an $\varepsilon$-cycle, then $s_1 \approx_{SD} s_2$. $\Box$

**Lemma 6** [Elor94] Let $K_i = (S_i, Act_\tau, T_i, s0_i)$, $i = 1, 2$, be two lts's, $t \in Act^*$, $s_1 \in S_1$, $s_2 \in S_2$, and $s_1 \approx_{SD} s_2$. Whenever $s_1 \stackrel{t}{\Longrightarrow} s_1'$ then, $\exists s_2' \in S_2 : s_2 \stackrel{t}{\Longrightarrow} s_2' \wedge s_1' \approx_{SD} s_2'$. $\Box$

The proofs in [Elor94] for the two lemmas above, given for D-bisimilarity, are directly transferable with obvious modifications to the case of SD-bisimilarity.

**Definition 7** Let $K = (S, Act_\tau, T, s0)$ be a finite lts.
The lts $K^{sc} = (S^{sc}, Act_\tau, T^{sc}, s0^{sc})$ is *state-condensed* from $K$ with respect to $\approx_{SD}$, written $K^{sc} = \mathbf{condense}(K, \approx_{SD})$, iff
$S^{sc} = \{s^{sc} \mid s^{sc}$ is an equivalence class of $\approx_{SD} \wedge \exists s \in s^{sc}, \exists t \in Act_\tau^* : s0 \stackrel{t}{\longrightarrow} s\}$, $s0 \in s0^{sc}$, and $T^{sc}$ is the set of transitions containing $(s_1^{sc}, a, s_2^{sc})$, where $s_1^{sc} \neq s_2^{sc}$ or $a \neq \tau$, iff $s_1 \stackrel{a}{\longrightarrow} s_2$ for some $s_1 \in s_1^{sc}$ and $s_2 \in s_2^{sc}$, and $(s^{sc}, \tau, s^{sc})$ iff $\exists s \in s^{sc} : sdiv(s)$. $\Box$

We got the definition for the *strict divergence bisimulation*, *$SD$-bisimulation*, by modifying the definition of D-bisimulation in [Elor94] by demanding $R \subseteq SD = \{(s_1, s_2) \in S_1 \times S_2 \mid sdiv(s_1) = sdiv(s_2)\}$ instead of $R \subseteq D = \{(s_1, s_2) \in S_1 \times S_2 \mid div(s_1) = div(s_1)\}$. This modification leads to our novel concept *strict divergence bisimilarity*, *SD-bisimilarity* ($\approx_{SD}$). It is not difficult to see that this is a slightly stronger equivalence than the D-bisimilarity ($\approx_D$). For finite lts's it satisfies the relation $\approx_{SD} \subseteq \approx_D$, i.e., if $K_1 \approx_{SD} K_2$ we can deduce

$K_1 \approx_D K_2$. This holds because for finite lts's $div(s) \equiv sdiv(s) \vee (\exists s' \in S : s \overset{\varepsilon}{\Longrightarrow} s' \wedge sdiv(s'))$ and thus $SD \subseteq D$. The lts $\mathbf{condense}(K, \approx_{SD})$ is state-minimal with respect to SD-bisimilarity. It is obvious that $K \approx_D \mathbf{condense}(K, \approx_D) \approx_D \mathbf{condense}(K, \approx_{SD}) \approx_{SD} K$.

# 3 Verifying a progress property

In a simple mutual exclusion algorithm two sequential algorithms co-operate communicating through some shared variables. In LOTOS specification language [BoBr87] this kind of system can be represented as a parallel composition[1]

$$G = P_1|[L_1]|Memory|[L_2]|P_2, \tag{1}$$

where $L_1 \subset Act(P_1)$, $L_2 \subset Act(P_2)$, $Act(P_1) \cap Act(P_2) = \emptyset$.

In general, we consider shared memory concurrent systems of the form

$$G = Memory|[L_1]|P_1|[L_2]|\cdots|[L_n]|P_n \tag{2}$$

and assume that $G$ has no $\tau$-transitions, $Act(Memory) \subset \bigcup_{i=1}^{n} Act(P_i) = Act(G)$ and $i \neq j \Rightarrow Act(P_i) \cap Act(P_j) = \emptyset$.

We do not allow invisible transitions in the system $G$. This restriction is adopted in order to make it clear from which component process $P_i$ a transition in $G$ originates. This information is needed when applying fairness constraints to execution sequences of $G$.

A question arises: When does a system specified in Basic LOTOS have an invisible $\tau$-transition in its lts? Looking at the transition definitions in LOTOS we see that invisible transitions are created only by $i$-prefixing, hiding and enabling. These operations should not thus be used in specifications of component processes $P_1, \cdots, P_n$ and *Memory*. This restriction does not limit us in any way while verifying liveness properties of shared memory mutual exclusion algorithms [Kuup2000].

Let us now consider an lts $G$ and try to figure out the conditions under which we can be sure that[2]

$$\Box(req_1 \rightarrow \Diamond enter_1) \tag{3}$$

[1] The composition formula for $G$ is associative due to the assumptions about the label sets $L_1$ and $L_2$. Thus we do not need any parentheses in expression (1). We assume that the reader knows the elements of Basic LOTOS.

[2] We mean with this stylized temporal logic notation that from any transition labeled $req_1$ there is no path in the global state graph where we would not eventually pass a transition labeled $enter_1$.

on any action trace through the system $G$. We are here thinking on a mutual exclusion algorithm. The action $req_1$ means ”$P_1$ requests an entry to the critical program part” and $enter_1$ means ”$P_1$ successfully enters the critical program part”. This is a ”leads-to” type of liveness condition: If the process $P_1$ requests an entry, it will always eventually be able to do it.

We assume in our developments below that the process $P_1$ (as well as any other process the progress properties of which we want to investigate) has the following regularity property: In any action trace of $P_1$ an $enter_1$-transition is never followed immediately by another $enter_1$-transition, but a string of (at least one) actions followed possibly by a new $req_1$-action. Only after a new $req_1$-action can an $enter_1$-action occur anew.

If there would be an action trace where after $req_1$ there would not come any $enter_1$ this would mean that, assuming the system does not deadlock, there would be an infinite trace without an $enter_1$-action. In a finite lts an infinite trace is possible only if the trace forms a cycle.

**Analysis of the system structure**

We consider a system $G$ describing the cooperation of $n$ processes $P_1, \cdots, P_n$ with a shared memory *Memory* as given above in (2). Let us assume that the state space of $G$ is finite.[3] We represent $G$ as a labeled transition system and remove from $G$ all outgoing transitions from the target states of $enter_1$-transitions and get a new restricted system graph $\hat{G}$. The states and transitions not reachable from the target states of $req_1$-transitions in $\hat{G}$ are not included in $\hat{G}$ and its state space $\hat{S}$.

The subgraph $\hat{G}$ of $G$ can be subdivided into $N$ maximal strongly connected components $G_i$ with state sets $S_i$ and transition sets $TR(G_i)$:

$$\hat{S} = \bigcup_{i=1}^{N} S_i, \text{where } i \neq j \to S_i \cap S_j = \emptyset.$$

The transitions $TR(\hat{G})$ of $\hat{G}$ which are not in the set of *cross-component trans-*

We phrase the condition, for simplicity, for the process $P_1$. Any other process, and any other pair of transition labels, could be chosen for progress analysis, too. The methods presented can also be generalized to a case where $req_1$ is replaced by a set of alternative progress initialization labels and $enter_1$ with a set of alternative progress termination labels.

[3] A straightforward application of Basic LOTOS transition rules would always create an infinite lts for a recursive LOTOS definition. However, for an inherently finite system it is possible to find a finite lts isomorphic with the infinite lts created by the recursive LOTOS definition.

*itions*, $CCTR$, can be subdivided similarly:

$$TR(\hat{G}) \setminus CCTR = \bigcup_{i=1}^{N} TR(G_i), \text{where } i \neq j \rightarrow TR(G_i) \cap TR(G_j) = \emptyset.$$

We call every maximal strong component $G_i \subset \hat{G}$ from which a cross-component transition leads out an *inner* component. If there is no transition leading out of a component $G_i$ in $\hat{G}$ we call it a *terminal* component. We see, due to how $\hat{G}$ is constructed, that an $enter_1$-transition always leads to an one-state terminal component of $\hat{G}$. Other kind of terminal components are possible, too: There could be, e.g., a deadlock or a live-lock in the system.

It is obvious that maximal strongly connected components of $\hat{G}$ have the following property: If there is a transition $tr$ leading out of a component $G_i$ there is within $\hat{G}$ no route starting with $tr$ which would lead back to the same component $G_i$. Otherwise the component $G_i$ would not be maximal.

### Strong fairness

We assume first *strong process fairness*: If a transition of $P_i$ is enabled an infinite number of times at an execution path in $G$ some transition of $P_i$ will eventually be executed.

We will see that a necessary and sufficient condition that guarantees the progress condition (3) at page 4, when strong process fairness is assumed, is the following:

**Condition 8** For any strongly connected subgraph $SSG$ of $G_i$, where $G_i$ is an inner maximal strong component of $\hat{G}$, there is a process $P_h$ no transition of which is included in $SSG$ and some transition of which is enabled at a state of $SSG$. All the terminal components in $\hat{G}$ are entered only by an $enter_1$-transition.

**Lemma 9** In a strongly fair execution sequence of a finite-state system $G$ for which the Condition 8 holds only a finite number of consecutive execution steps can take place within any inner maximal strong component.

**Proof:** If there would be an infinite number of consecutive execution steps within an inner maximal strong component $G_i$, which is assumed to be finite, the states and transitions belonging an infinite number of times to this execution path would form a strongly connected subgraph $SSG \subseteq G_i$. According to the

Condition 8 there is a process $P_h$, no transition of which is included in $SSG$ and some transition of which is enabled at a state of $SSG$. Due to the strong fairness assumption some transition of $P_h$ would eventually be taken at this execution path. This contradicts with the definition of $SSG$ used in this proof; there cannot be any infinite execution path within a $G_i$ fulfilling the Condition 8. □

Let us now consider a strongly fair execution sequence within $G$. We assume that the Condition 8 holds. According to Lemma 9 only a finite number of consecutive execution steps can take place within any inner strong component of $\hat{G}$. Once an execution path has left an inner maximal strong component, the execution path cannot return to this component again. Otherwise the component would not be maximal. Because we assume that the state space $\hat{G}$ is finite, the execution must eventually reach a terminal component of $\hat{G}$; due to the second sentence in the Condition 8 this must happen with an $enter_1$-transition. We can thus state the following proposition:

**Proposition 10** Any strongly fair execution sequence of the system $G$ executing a $req_1$-transition will eventually execute an $enter_1$-transition iff the Condition 8 holds.

**Proof:** The sufficiency of the Condition 8 was argued in the text preceding the Proposition 10. The necessity can be argued as follows.

Assume there would be a strongly connected subgraph $SSG \subseteq G_i$ for which the first sentence of the Condition 8 would not hold, i.e., for any process $P_h$, $i = 1, \cdots, n$, the following would hold: A transition of the process $P_h$ is either included in $SSG$ or no transition of $P_h$ is enabled in any $SSG$ state. Let the processes $\{P_h | h \in H\}$ be the ones which have transitions included in $SSG$ and consider an execution path containing an infinite number of times all transitions and states of $SSG$. No transition of the rest of the processes $\{P_k | k \notin H\}$ is enabled in any state of $SSG$. This execution path is clearly strongly fair and stays forever within $SSG$ without ever reaching an $enter_1$-transition.

If there would be a terminal component of $\hat{G}$ which would be entered by other transitions than $enter_1$, there are two possibilities. (1) The terminal component of $\hat{G}$ is a single state of $G$ without any transitions where an $enter_1$-transition leads into. Thus there would be a route in $G$ to pass by the $enter_1$-transition. We could now find an execution not containing an $enter_1$-transition before the possible next $req_1$-transition. (2) The terminal component of $\hat{G}$ does not contain the target state of any $enter_1$-transition. Then this would form a set of states and transitions where an execution path could stay forever (or stop) without ever reaching an

$enter_1$-transition. Any $enter_1$-transition would lead out of the component and this is not an inner component. $\Box$

The Condition 8 can be checked algorithmically:

(1) Form the maximal strong components of $\hat{G}$ and determine its terminal components.

(2) Check that the Condition 8 holds for all components of $\hat{G}$.

(3) If the Condition 8 does not hold for a component of $\hat{G}$ we are able to give a strongly fair execution of $G$ where the condition (3) at page 4 does not hold. This would either lead to a terminal component, without executing an $enter_1$-transition, or stay within an inner component forever, if the first sentence of the Condition 8 would not hold. $\Box$

The test in step (2) above whether the first sentence of the Condition 8 holds (excluding the trivial test for the terminal components) can be done by calling $Live(G_i)$ using the following test function:

$Live(SSG)$: (Test a strongly connected subgraph $SSG$; test gives either $\mathtt{true}$ or **false** according to whether the Condition 8 holds or not.)

1. Let $I$ and $J$ be two sets of indices: $I = \{i \mid$ a transition of $P_i$ is included in $SSG\}$, $J = \{i \mid$ a transition of $P_i$ is enabled at some state of $SSG\}$.

2a. If $I = \emptyset$ return $Live(SSG) = \mathtt{true}$ (there are no transitions in $SSG$).

2b. If $I = J \neq \emptyset$, return $Live(SSG) =$ **false** (there is no such $P_h$ required in Condition 8 and thus there is a fair execution which stays forever in $SSG$).

3. If $I \neq J$ ($I \subset J$), do for each $h \in J \setminus I$ the following: Remove from $SSG$ all the states where a transition of $P_h$ is enabled. The resulting graph $SSG^*$ has all the remaining states of $SSG$ and all transitions whose start and target states are included in $SSG^*$. (If there is a subgraph of $SSG$ which contains a fair execution it cannot contain the initial state of any $P_h$-transition, $h \in J \setminus I$.)

4. Form the following set of strongly connected subgraphs of $SSG$: $R = \{SSG' \mid SSG'$ is a maximal strong component of $SSG^* \wedge SSG'$ has at least one transition $\}$. (If there is a fair execution within $SSG$ it must be included in one of the subgraphs of $R$.)

If the test $Live(SSG')$ gives the value **false** for any graph $SSG'$ in this set $R$, return $Live(SSG) =$ **false** (there is a fair execution staying forever in an $SSG'$). Otherwise (if all tests $Live(SSG')$ return with value $\mathtt{true}$) or if $R$ is an empty set, return $Live(SSG) = \mathtt{true}$. $\Box$

The recursion terminates in $Live(SSG)$ because at each recursion step at least one node is removed from the finite graph $SSG$.

**Weak fairness**

We consider now weakly fair execution sequences of the system $G$ and study whether the progress condition (3) on page 4 would hold. Weak process fairness means here that if a transition of a process $P_i$ is continually enabled after a finite number of steps at an execution path some transition of $P_i$ will eventually be executed.

Let us assume that the following condition holds for any strong component of $\hat{G}$:

**Condition 11** For any inner maximal strong component $G_i \subset \hat{G}$ there is a process $P_h$ any transition of which is not included in $G_i$ and some transition of which is enabled at all the states of $G_i$. All the terminal components in $\hat{G}$ are entered only by $enter_1$-transitions.

We are now able to see that the following lemma holds:

**Lemma 12** Let us consider a weakly fair infinite execution sequence of a finite-state system $G$. Assume that the Condition 11 holds. Then only a finite number of consecutive execution steps can take place within any inner component $G_i$.

**Proof:** If there would be an infinite number of consecutive execution steps within an inner component $G_i$, this would conflict with the weak fairness assumption because the Condition 11 is assumed to hold. Some transition of the process $P_h$, mentioned in the Condition 11, would be all the time enabled and thus eventually executed leading the execution path out from $G_i$. $\Box$

With the help of the Lemma 12, in the similar way we argued for Proposition 10 in case of strong fairness, we can see that for a weakly fair system the progress condition (3) at page 4 is valid iff Condition 11 holds. Thus we see that the following proposition holds:

**Proposition 13** Any weakly fair execution path of the system $G$ passing a $req_1$-transition will eventually pass an $enter_1$-transition, i.e., the condition (3) at the page 4 holds, iff the Condition 11 holds.

We also see a straightforward algorithmic method to check whether the progress condition (3) at page 4 holds:

(1) Form the maximal strong components of $\hat{G}$ and find the terminal components.

(2) Check that the Condition 11 holds. If it holds, the progress condition (3) holds, too.

(3) If the Condition 11 does not hold for a component $G_i$ of $\hat{G}$ we are able to give a weakly fair execution sequence of $G$ where the progress condition (3) does not hold: This sequence would either lead to a terminal component without an $enter_1$-transition, or stay within an inner component forever, if the first sentence of Condition 11 would not hold. $\Box$

The test function $Live(SSG)$ given at page 8 can be modified to test with the call $Live(G_i)$ the Condition 11 for the component $G_i$. Modify (and simplify) step 3 to the form: If there is an index $h \in J \setminus I$ such that in all states of $SSG$ a transition of $P_h$ is enabled, return $Live(SSG) = \mathbf{true}$. Otherwise return $Live(SSG) = \mathbf{false}$ (there is a fair execution which stays forever in $SSG$). Step 4 is omitted; there is no recursion in this modified algorithm.

A way to guarantee that a transition in a participating process $P_h$ is always enabled in $G$, as asked in the Condition 11, is to find out that the system has the following property, to be called *all-activeness*:

**Condition 14** At any state of a $n$ process system $G$ all the processes $P_1, \cdots, P_n$ are enabled.

When specifying a shared memory concurrent system it is not difficult to achieve all-activeness, because *Memory* is always ready to make transitions together with any active component process $P_i$ and there are no transitions where more than one of the processes $P_1, \cdots, P_n$ would participate.

All-activeness is simple to check algorithmically in a global state graph $G$ when $G$ does not have any $\tau$-transitions and its component processes $P_1, \cdots, P_n$ have different action alphabets, as was assumed for our system $G$. We simply check that from any state of $G$ there are the transitions labeled with $a_i \in Act(P_i)$, $i = 1, \cdots, n$, i.e., at least one transition of each participating process $P_i$.

For an all-active system the Condition 11 is logically equivalent with the following:

**Condition 15** $G$ is all-active. Transitions of at least one process are not included in any inner maximal strong component of $\hat{G}$. All the terminal components in $\hat{G}$ are entered only by $enter_1$-transitions.

### Generalizations

In our approach we assume a shared memory system where participating active processes $P_i$ use different action alphabets, sharing their alphabet only with the $Memory$-component. We also disallow the use of any invisible transitions labeled with $\tau$ in the process descriptions. These restrictions were necessary to make it obvious from which process any transition in $G$ originates.

If we would want not to have these restrictions, the software tool generating the global state graph $G$ should mark in $G$ each transition with a flag indicating which process has performed this transition. If there would be no rendez-vous synchronizations between participating processes this would allow us to use the progress checking methods given in this section.

If there would be also rendez-vous synchronizations between the processes some modifications would be needed. A transition in $G$ resulting from a rendez-vous synchronization between processes $\{P_i | i \in R\}$ should be counted as a transition of all these processes and also, when executed, contributing to the fairness requirements concerning all the processes $\{P_i | i \in R\}$. Thus the process sets in the test function $Live(SSG)$ should now include all participating processes: $R \subseteq I$ if the synchronization transition is included in $SSG$; $R \subseteq J$ if the synchronization transition is enabled at a state in $SSG$ (possibly leading out of $SSG$).

We have used process fairness (like in [LiPn84]) in formulating our conditions and algorithms. If we would want to employ transition oriented versions of strong and weak fairness we could easily reformulate the conditions 8 and 11 for this case. The definition of strong fairness would now be: "Consider a transition $t$ in a given set $\mathcal{C}$ of transitions selected from the participating processes. If $t$ is enabled an infinite number of times $t$ will be eventually be taken." The transition oriented version of weak fairness would be defined analogously.

Now the first sentence of the Condition 8 should be written: "For any strongly connected subgraph $SSG$ of $G_i$, where $G_i$ is an inner maximal strong component of $\hat{G}$, there is a transition $t \in \mathcal{C}$ which is not included in $SSG$ and which is enabled at a state of $SSG$." Similarly the Condition 11 should be rephrased to the form:"For any inner maximal strong component $G_i \subset \hat{G}$ there is a transition $t \in \mathcal{C}$ which is not included in $G_i$ and which is enabled at all states of $G_i$." The lemmas 9 and 12 as well as propositions 10 and 13 would continue to hold. Also the test function $Live(SSG)$ can be modified to handle transition oriented fairness. Instead of index sets $I$ and $J$ we should consider analogous transitions sets $\mathcal{I} \subseteq \mathcal{C}$ and $\mathcal{J} \subseteq \mathcal{C}$ in the modified $Live(SSG)$-function.

Thus it looks like a general-purpose process-algebraic tool for progress checking could be constructed based on the algorithms and conditions given in this section.

# 4 Deducing progress properties from a state-condensed system

We want now to study whether it is possible to deduce something of the progress properties of a system $G$ by looking at a condensed system after hiding part of the action labels. This turns out to be possible when weak fairness and an all-active system $G$ are assumed. For strong fairness we have not been able to find any corresponding method.

We present first three general propositions about minimal systems to help us in our deductions.

**Proposition 16** Let us consider a labeled transition system $G$ and let $A \subseteq Act(G)$ be a set of visible actions such that $req_1 \notin A$ and $enter_1 \notin A$.

Let[4] $H = \textbf{hide}\, A\, \textbf{in}\, G$, $B = Act(G) \setminus A$ and $M = \textbf{condense}(\approx_{SD}, H)$. The subsystems $\hat{G}$, $\hat{H}$ and $\hat{M}$ are gained from $G$, $H$ and $M$ by removing the outgoing transitions from the target states of $enter_1$-transitions and by removing the states unreachable from $req_1$-transitions.

Then,

(a) there is an isomorphic bijection $f(s)$ between the states of $G$ and $H$ as well as between the transitions of $G$ and $H$ such that a transition $(s, a, s') \in G$ is mapped to a transition $f((s, a, s')) = (f(s), a', f(s')) \in H$ where $a' = a$ if $a \in B$, otherwise $a' = \tau$.

(b) The strongly connected subgraphs (*sc-subgraphs*) with respect to reachability relation in $G$ and $H$ are bijectively mapped to each other.

(c) There is a surjection $h(s)$ induced by **condense**-function from the states of $H$ to the states of $M$. There is a transition $(m, a, m') \in M$, $a \in B$, iff there is a transition $(s, a, s') \in H$ such that $h(s) = m$ and $h(s') = m'$. There is a transition

[4] The operation **hide** results an lts $H$ otherwise identical with $G$ but all the transition labels $\in A$ are replaced in $H$ with $\tau$.

$(m, \tau, m') \in M$ only if there is a transition $(s, \tau, s') \in H$ such that $h(s) = m$ and $h(s') = m'$.
(d) The sc-subgraphs in $H$ are injectively mapped to the sc-subgraphs in $M$.
(e) $\hat{H} \approx_{SD} \hat{M}$
(f) For any maximal strongly connected subgraph (*msc-subgraph* or *strong component*) $M'$ of $\hat{M}$ there is a msc-subgraph $H'$ of $\hat{H}$ which is mapped to $M'$ by the **condense**-function.
(g) Any terminal msc-graph $H' \in \hat{H}$ is mapped to a terminal msc-graph $M' \in \hat{M}$.
(h) Any terminal component entered by an $enter_1$-transition in $\hat{H}$ is mapped to a terminal component entered by an $enter_1$-transition in $\hat{M}$. $\Box$

**Proof:** (a) Follows directly from the definitions. (In fact, if we do not want to rename the $H$-states, $f(s) = s$.)

(b) Due to (a) $s'$ is reachable from $s$ iff $f(s')$ is reachable from $f(s)$. Thus $G'$ is an sc-subgraph in $G$ iff $f(G')$ is an sc-subgraph in $H$.

(c) Follows directly from the definition of the **condense**-function.

(d) Any sc-subgraph can be represented as a closed cycle of states and transitions, beginning and ending at the same state. This cycle might visit some states and transitions several times. Any such cycle represents an sc-subgraph.

Let $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_n$, $s_n = s_1$, $n \geq 1$, be a cycle in $H$ representing an sc-subgraph. We can construct in $M$ a corresponding cycle $h(s_1) \rightarrow \cdots \rightarrow h(s_n)$ as follows: If $a_i \neq \tau \vee s_i \not\approx s_{i+1}$ we can deduce from the definition of the **condense**-function that $h(s_i) \xrightarrow{a_i} h(s_{i+1})$ in $M_1$. This transition is taken to the image cycle. The same holds if $a_i = \tau \wedge s_i \approx s_{i+1} \wedge sdiv(s_i)$. If $a_i = \tau \wedge s_i \approx s_{i+1} \wedge \neg sdiv(s_i)$ it is obvious that $h(s_i) = h(s_{i+1})$ and the transition $h(s_i) \xrightarrow{a_i} h(s_{i+1})$ is not taken to the image cycle in $M_1$.

The created transition sequence is a cycle in $M$, $h(s_1) = h(s_n)$, and contains the states $\{h(s) \mid s \in \{s_1, \cdots, s_n\}\}$ and a set of $M$-transitions. The mapping from an arbitrary sc-subgraph of $H$ to an sc-subgraph of $M$ is thus established and the claim (d) is thus proved.

(e) Let $R_1 = \{(h, m) | h \approx_{SD} m\}$ be a bisimulation between systems $H$ and $M$. Then $R_2 = \{(h, m) | (h, m) \in R_1 \wedge h \in \hat{H} \wedge m \in \hat{M}\} \cup \{(h_0, m_0)\}$, where $h_0$ and $m_0$ are the start states of the systems $\hat{H}$ and $\hat{M}$ (having only outgoing $req_1$-transitions), is a bisimulation between $\hat{H}$ and $\hat{M}$. The bisimulation $R_2$ compares the states that can be reached with an action string of the form $req_1 any^*$ and $req_1 any^* enter_1$, where $any$ is any action, $enter_1$ excluded.

(f) Let $m_1 \xrightarrow{a_1} m_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{k-1}} m_k$, $m_1 = m_k$, $k \geq 1$, be a cycle representing an msc-subgraph $M'$ in $\hat{M}$. Can we always find an msc-subgraph $H'$ in $\hat{H}$ which is mapped to this subgraph in $\hat{M}$ by the construction given in (d) above?

Obviously, because $\hat{H} \approx \hat{M}$, due to the Lemma 6 there is in $\hat{H}$ a sequence of weak transitions $s_1^{(1)} \stackrel{a_1}{\Longrightarrow} s_2^{(1)} \stackrel{a_2}{\Longrightarrow} \cdots \stackrel{a_{k-1}}{\Longrightarrow} s_k^{(1)}$ such that $h(s_i^{(1)}) = m_i$, $i = 1, \cdots, k$. But we cannot be sure that $s_1^{(1)} = s_k^{(1)}$ although we know that $s_1^{(1)} \approx s_k^{(1)}$. Therefore, we continue by constructing a second transition sequence $s_k^{(1)} = s_1^{(2)} \stackrel{a_1}{\Longrightarrow} s_2^{(2)} \stackrel{a_2}{\Longrightarrow} \cdots \stackrel{a_{k-1}}{\Longrightarrow} s_k^{(2)}$, a third transition sequence etc. We know that $m_1 \approx s_1^{(1)} \approx s_1^{(2)} \approx \cdots$. Because each equivalence class contains a finite number of $\hat{H}$-states there must be two indices $h_1$ and $h_2$ such that $h_1 < h_2$ and $s_1^{(h_1)} = s_1^{(h_2)}$. Then we have in $\hat{H}$ a closed cycle $C$, $s_1^{(h_1)} \stackrel{a_1}{\Longrightarrow} \cdots \stackrel{a_{k-1}}{\Longrightarrow} s_k^{(h_1)} = s_1^{(h_1+1)} \stackrel{a_1}{\Longrightarrow} \cdots \stackrel{a_{k-1}}{\Longrightarrow} s_k^{(h_2-1)} = s_1^{(h_2)}$ which represents an sc-subgraph in $\hat{H}$ and which is mapped to a cycle in $\hat{M}$.

But in a transition sequence $s_i^{(h)} \stackrel{a_i}{\Longrightarrow} s_{i+1}^{(h)}$ there might appear additional intermediate states which are $\not\approx m_i$ and $\not\approx m_{i+1}$, or $C$ could still need to be expanded in some other way to form a maximal sc-subgraph $C'$, $C \subseteq C'$, of $\hat{H}$. Due to the claim (d) above this sc-subgraph $C'$ is mapped to an sc-subgraph in $\hat{M}$ which could possibly be larger than the original sc-subgraph $M'$ and contain $M'$ as its subgraph. But $M'$ is assumed to be maximal and thus this is impossible. Clearly the msc-subgraph $C'$ in $\hat{H}$ must be mapped to the msc-subgraph $M'$ in $\hat{M}$.

(g) Let $s \in H'$, thus $h(s) \in M'$. If $M'$ would not be a terminal component there would be a transition route $h(s) \stackrel{t}{\Longrightarrow} h(s')$ such that $h(s') \in \hat{M} \wedge h(s') \notin M'$ we would have, due to Lemma 6, a state $s'' \in \hat{H}$ such that $s \stackrel{t}{\Longrightarrow} s'' \wedge s'' \approx_{SD} h(s')$. Because $h(s'') = h(s') \notin M'$, we conclude $s'' \notin H'$ and thus $H'$ could not be terminal.

(h) Let $H' \in \hat{H}$ be a terminal component entered by an $enter_1$-transition. According to the definition of $\hat{H}$ this terminal component has only one state and no transitions. As $\hat{M} \approx_{SD} \hat{H}$ there must be a state $m \in \hat{M}$ bisimilar with the only state in $H'$, this state must also be entered by $enter_1$-transition and there cannot be any transitions out of $m$. $\Box$

**Proposition 17** We consider a system $G$ with the assumptions and notation introduced in Proposition 16. Then for the following classes of msc-subgraphs there is in $\hat{H}$ a msc-subgraph mapped by **condense**-function to a msc-subgraph of the same class in $\hat{M}$:

(a) transition-less msc-subgraphs (no transitions within the component)

(b) invisible msc-subgraphs (there are no visible transitions, only invisible $\tau$-transitions within the msc-subgraph)

(c) partially visible msc-subgraphs (there are both visible $Act(P_1)$-transitions and invisible $\tau$-transitions in the msc-subgraph).

**Proof:** (a) When there is only one state $m$ and no transitions in an msc-subgraph of $\hat{M}$ there cannot be any transitions in the msc-subgraph in $\hat{H}$ mapped to $m$ as given in Proposition 16: Otherwise there would be a $\tau$-loop or a visible $B$-loop at $m$ due to the definition of the **condense**-function.

(b) If there are only $\tau$-transitions in an msc-subgraph of $\hat{H}$, due to Lemma 5 and the definition of **condense**-function, there is only one state and one $\tau$-transition (a $\tau$-loop) in the corresponding sc-subgraph of $\hat{M}$.

If there are only $\tau$-transitions in an msc-subgraph $M'$ of $\hat{M}$ it must be, due to the Lemma 5, a one-state subgraph with a $\tau$-loop. If there would be a visible $B$-transition in an msc-subgraph in $\hat{H}$ mapped to $M'$ there would be a visible $B$-transition in $M'$.

(c) Let $M'$ be a msc-subgraph in $\hat{M}$ and $H'$ a corresponding msc-subgraph in $\hat{H}$. Due to Proposition 16(c) there is a visible $B$-transition within $H'$ iff there is a visible $B$-transition within $M'$.

If there is a $\tau$-transition within $M'$ we can deduce from the definition of **condense**-function, due to the minimality of $M$, that there is a $\tau$-transition within $H'$, too. $\Box$

We cannot include the class of wholly visible msc-subgraphs (there are only visible $Act(P_1)$-transitions and no $\tau$-transitions in the msc-subgraph) in the Proposition 17, because there might be $\tau$-transitions in a subgraph of $\hat{H}$ but no $\tau$-transitions in the corresponding condensed subgraph of $\hat{M}$. There is, however, a useful special case which can be handled by the following proposition. It is presented for a general system size although it actually seems to be useful only for two process systems, i.e., when $n = 2$.

**Proposition 18** Let us consider a shared-memory concurrent system

$$G = Memory|[L_1]|P_1|[L_2]|P_2| \cdots |[L_n]|P_n$$

and assume that $G$ has no $\tau$-transitions. We assume that $Act(Memory) \subset \bigcup_{i=1}^{n} Act(P_i) = Act(G)$, and $i \neq j \Rightarrow Act(P_i) \cap Act(P_j) = \emptyset$.

Let $H = \textbf{hide } Act(G) \setminus Act(P_1) \textbf{ in } G$, $M = \textbf{condense}(\approx_{SD}, H)$ and the subsystems $\hat{H}$ and $\hat{M}$ be defined as in Proposition 16 above.

Let us assume we have an msc-subgraph $M'$ of $\hat{M}$ with only $Act(P_1)$-transitions, i.e., there appears no $\tau$-transitions within this msc-subgraph. We assume further that the $Act(P_1)$-actions of $M'$ do not change the state of the *Memory*-component in $G$. Then any sc-subgraph $H'$ in $\hat{H}$ mapped to the subgraph $M'$ has only $Act(P_1)$-transitions, i.e., there appear no $\tau$-transitions within $H'$. □

**Proof:** Let us assume that there would be a sc-subgraph $H''$ in $\hat{H}$ which would have both $\tau$-transitions and $Act(P_1)$-transitions and would be mapped by **condense**-function to the msc-subgraph $M'$ in $\hat{M}$. Let us write in LOTOS the system formula for $H$ as follows: $H = P_1|[L_1]|H_2$, where $H_2 = \mathbf{hide}\ Act(G) \setminus Act(P_1)\ \mathbf{in}\ Memory|[L_2]|P_2 \cdots |[L_n]|P_n$. Due to the process-algebraic compositionality we have $M = \mathbf{condense}(\approx_{SD}, P_1|[L_1]|M_2)$, where $M_2 = \mathbf{condense}(\approx_{SD}, H_2)$.

Let a cycle $C$ represent the sc-subgraph $H''$ in $H$. The cycle $C$ would consist of transitions of the form $s' \xrightarrow{a} s''$, $a \in Act(P_1)$, where the process $P_1$ moves alone or together with the subsystem $H_2$, and transitions of the form $s' \xrightarrow{\tau} s''$, where only the subsystem $H_2$ moves. The transitions $s' \xrightarrow{a} s'', a \in Act(P_1)$, do not change the state of the *Memory*-part and thus neither the state of $H_2$.

All the transitions $s' \xrightarrow{\tau} s''$ in the cycle $C$ correspond to $\tau$-transitions in the lts representing $H_2$. These $\tau$-transitions form a cycle. Thus there is an $\tau$-cycle in $H_2$ determined by the cycle $C$. In the composition $H = P_1|[L_1]|H_2$ and the subsequent condensation $M = \mathbf{condense}(\approx_{SD}, H)$ this cycle would form a $\tau$-loop at a state of $H''$ and hence also in $M$ and, because $M'$ was assumed to be an msc-subgraph of $\hat{M}$, also a $\tau$-transition to $M'$. This is a contradiction and thus the claim of the proposition holds. □

### Weak fairness

Let us consider now how we could deduce, assuming weak process fairness, that the Condition 11 for $G$ holds or does not hold by looking at the system $\hat{M}$ defined in Proposition 18. It turns out that this is possible when $n = 2$ but does not appear possible for $n > 2$. When $G$ has only two processes, $n = 2$, we call an all-active system also synonymously *biactive*.

We phrase first a condition for $H$ which for biactive systems, $n = 2$, is logically equivalent with the Condition 11 for system $G$. This can be seen by looking at Proposition 16 and Condition 15 which is logically equivalent with the Condition 11 in this case.

**Condition 19** In $\hat{H}$ no reachable inner strong component has both $\tau$- and $Act(P_1)$-transitions. All the terminal components in $\hat{H}$ can be entered only by $enter_1$-transitions.

**Proposition 20** We consider a biactive system $G$, $n = 2$, with the assumptions and notation given in Proposition 18. The Condition 19 holds for the system $H$ iff the Condition 11 holds for the corresponding system $G$.

Further, with the help of propositions 13, 16, 17, 18, and 20 we see that the following two propositions hold:

**Proposition 21** We consider a system $G$ defined in Proposition 20. If there is a reachable strong component in $\hat{M}$ having both $\tau$- and $Act(P_1)$-transitions or in $\hat{M}$ a terminal component is entered by other transitions than $enter_1$-transitions, the Condition 11 is violated and the system $G$ is not live, i.e., the progress condition (3) at page 4 does not hold under weak fairness assumption.

**Proposition 22** We consider a system $G$ defined in Proposition 20. If all the strong components in $\hat{M}$ have
(a) only one state and no transitions,
(b) only $\tau$-transitions, or
(c) only visible transitions which do not change the state of the *Memory*-components,
and all the terminal components of $\hat{M}$ are entered only by $enter_1$-transitions, the Condition 19 holds, and the system $G$ is live under weak fairness, i.e., (3) at page 4 holds in $G$.

**Remark 23** When $n = 2$ we can decide looking at the graph $\hat{M}$, based on propositions 21 and 22, whether the progress property (3) at page 4 holds in $G$ or not. Only in the case that the assumption (c) in Proposition 22 does not hold, i.e., there is a strong component $M'$ in $\hat{M}$ having only visible $Act(P_1)$-transitions and some of these do change the state of the *Memory*-component, we would be in doubt whether the system $G$ is live or not under weak fairness. We have not so far met in our case studies a situation like this. If that would ever occur we should look at the part of $\hat{H}$ corresponding to $M'$ and find out whether the Condition 19 holds or not.

In propositions 21 and 22 we assume a biactive system. This assumption can obviously be relaxed a little: the states belonging to transition-less msc-components in $G$ need not be biactive. Elsewhere the biactiveness assumption is needed as otherwise after condensation with respect to SD-bisimilarity, e.g., the lts's (a) and (b) in Figure 1 would give the same condensation result (c). However, under weak fairness the two systems (a) and (b), where $a$ represents an action in $Act(P_1)$ and $\tau$ a hidden $Act(P_2)$-action, would behave differently. No infinite weakly fair execution could stay forever in system (a), but that would be possible in the case of system (b).

It might appear at the first sight that the above condensation would be adequate when strong fairness is assumed. However, as can be seen from the lts (d) in Figure 1, which also condenses to (c), that this is not necessarily the case. The system represented by lts (d) is not live under strong fairness.

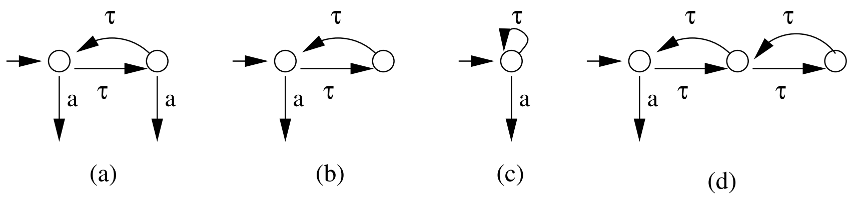


Figure 1: Four different SD-bisimilar strong components

# 5 Case studies: Peterson's and Lamport's mutual exclusion algorithms.

As a demonstration of the approach given above to check progress properties of a shared memory parallel algorithm, we consider the well-known Peterson's algorithm [Andr91] for mutual exclusion of two processes. This example is from [Kuup2000] where full details, and also further case studies, can be found. The cooperating processes $P_1$ and $P_2$ share two Boolean variables $b_1$ and $b_2$, which are initially **false**, and an integer variable $k$ and have the mutual exclusion property. The program fragment given in Figure 2 defines $P_1$ when $i = 1$ and $j = 2$, and $P_2$ when $i = 2$ and $j = 1$.

We have added signals $req_i$, $enter_i$, and $exit_i$ to the program to indicate the events "$P_i$ requests entry to its critical section", "$P_i$ enters its critical section", and "$P_i$ exits its critical sections".

We have used Basic LOTOS to model this system. Because Basic LOTOS does not have any variables each variable must be described by a process having separate states for separate variable values and access events for reading and writing variable values. Due to our theoretical developments for liveness checking we

**while true do**
**begin**
  non-critical section
  **signal**$(req_i)$
  $b_i :=$ **true**
  $k := j$
  **while**$(b_j \wedge k = j)$ **do skip**
  **signal**$(enter_i)$
  critical section
  **signal**$(exit_i)$
  $b_i :=$ **false**
**end**

Figure 2: Peterson's algorithm for mutual exclusion of two processes.

use separate access events for each process accessing the variables. The lts of the process $B_1$ describing the boolean variable $b_1$ is given in Figure 3. The coding of variable access events is interpreted as follows: *b1r1f* means "process $P_1$ reads the value **false** from the variable $b_1$", similarly *b1w2t* means "the process $P_2$ writes to the variable $b_1$ the value **true**". The whole memory component in the formula (1) can now be composed as $Memory = B_1|||B_2|||K$, where $B1$, $B2$, and $K$ are the processes representing the variables $b_1$, $b_2$, and $k$.

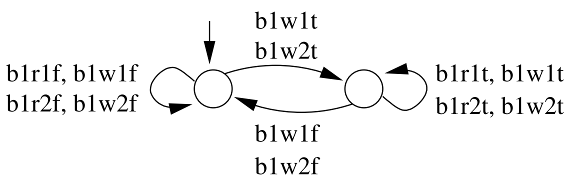


Figure 3: The lts of the process $B_1$ describing access to the variable $b_1$.

The processes $P_1$ and $P_2$ of the Peterson's algorithm can be written in Basic LOTOS and converted to lts's. With the memory access coding explained above the lts corresponding to the process $P_1$ is given in Figure 4. Inspired by the argumentation of a similar case study in [VaSe96] we have added an $idle1$-transition to the start state in order to model the fact that the process $P_1$ may stay in its non-critical section forever without ever executing a $req_1$-transition.

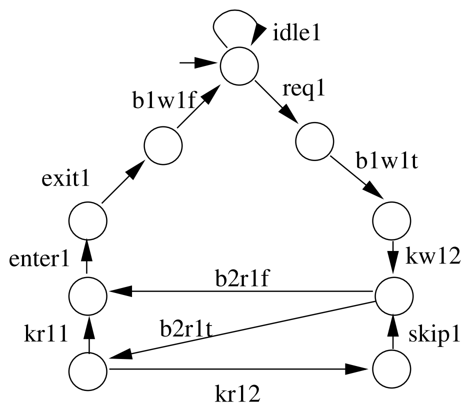


Figure 4: The lts of the process $P_1$ in Peterson's algorithm.

In order to investigate whether the process $P_1$ always eventually enters its critical section we form the systems $G$, $H$, and $M$ defined in Proposition 18. The system $G$ is biactive, as needed by our liveness verification method. The systems $G$ and $H$ have 94 states and 216 transitions. The condensed system $\hat{M}$ has 29 states and 65 transitions.

It is now possible to conclude the liveness of the process $P_1$ in Peterson's algorithm, by Remark 23: We first notice that there are no cycles with both visible and invisible events in $\hat{M}$. In fact, the only maximal strong components of $\hat{M}$ which have more than one state are the components formed by the state sets $\{5, 6, 7\}$ and $\{11, 12, 13\}$. Further, the terminal components of $\hat{M}$ are the one-state components $\{3\}$, $\{30\}$ and $\{31\}$ and these are entered only with $enter_1$-transitions in $\hat{M}$. We should still look at the Figure 5, where $\hat{M}$ is displayed, whether $\hat{M}$ contains any cycles with visible events. We notice that there are two cycles of this type: $12 \stackrel{\text{b2r1t}}{\longrightarrow} 11 \stackrel{\text{kr12}}{\longrightarrow} 13 \stackrel{\text{skip1}}{\longrightarrow} 12$ and $5 \stackrel{\text{b2r1t}}{\longrightarrow} 7 \stackrel{\text{kr12}}{\longrightarrow} 6 \stackrel{\text{skip1}}{\longrightarrow} 5$. Here kr12 and b2r1t are reading events, skip1 is a dummy control event, which do not change the memory state. Thus the assumptions of the Proposition 22 hold and $P_1$ is live, i.e., when it signals $req_1$ it is able to signal $enter_1$, too, and thus it always eventually reaches its critical program section.

We also studied Lamport's algorithm for the mutual exclusion of two pro-

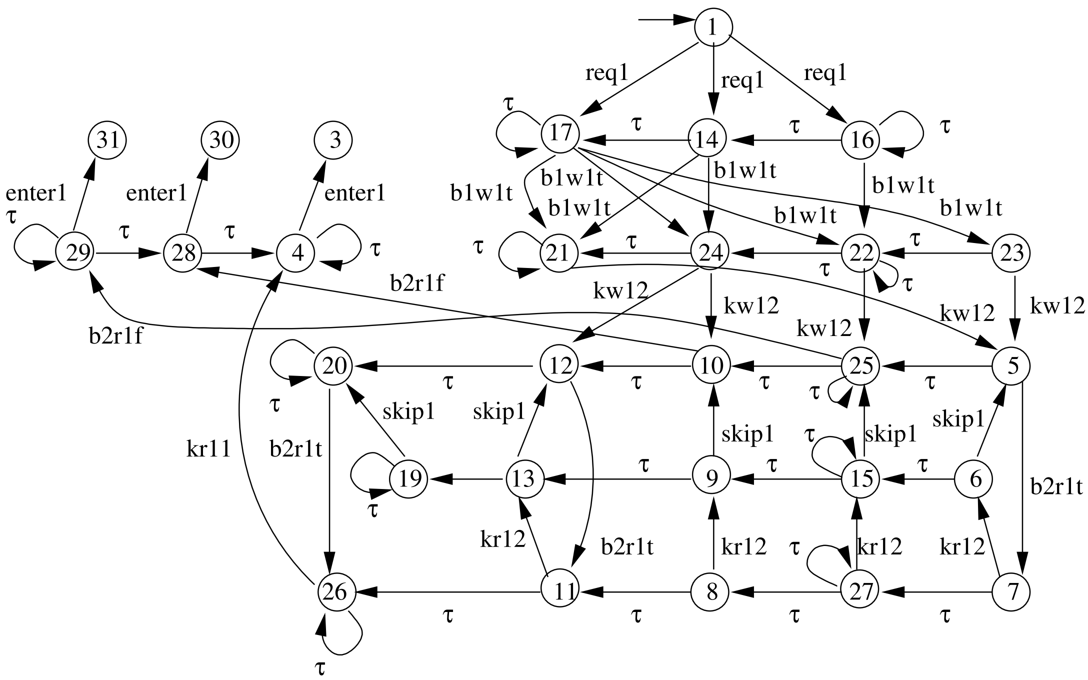


Figure 5: The lts $\hat{M}$ in the case of Peterson's algorithm

cesses [Walk89], [Lamp86]. This is known to be live for process $P_1$ and not live for process $P_2$. The condensed state graphs, based on a biactive global state graph $G$, needed by Remark 23 to check their progress properties are given in figures 6 and 7. The Figure 6 confirms the fact that Lamport's process $P_1$ always eventually reaches the $enter_1$-transition and enters its critical program part: In $\hat{M}$ there are the visible cycles 19 $\stackrel{\text{skip1}}{\longrightarrow}$ 20 $\stackrel{\text{b2r1t}}{\longrightarrow}$ 19 and 13 $\stackrel{\text{b2r1t}}{\longrightarrow}$ 14 $\stackrel{\text{skip1}}{\longrightarrow}$ 13, but the actions in these cycles do not change the memory state. The terminal components $\{2\}$, $\{22\}$ and $\{23\}$ in $\hat{M}$ are entered only by $enter_1$-transitions, and there are no cycles in $\hat{M}$ with both visible and invisible transitions.

In Figure 7 there is a reachable strong component in $\hat{M}$ consisting of states $\{4, 5, 6, 7, 8, 9, 10, 11, 12, 18, 19, 20, 21\}$ and containing both visible and invisible transitions. One can deduce by Proposition 21 that after $P_2$ has requested entry to the critical program part Lamport's algorithm does not guarantee its entry under weak fairness, i.e., it is possible that $P_2$ is never able to perform an $entry_2$ transition.

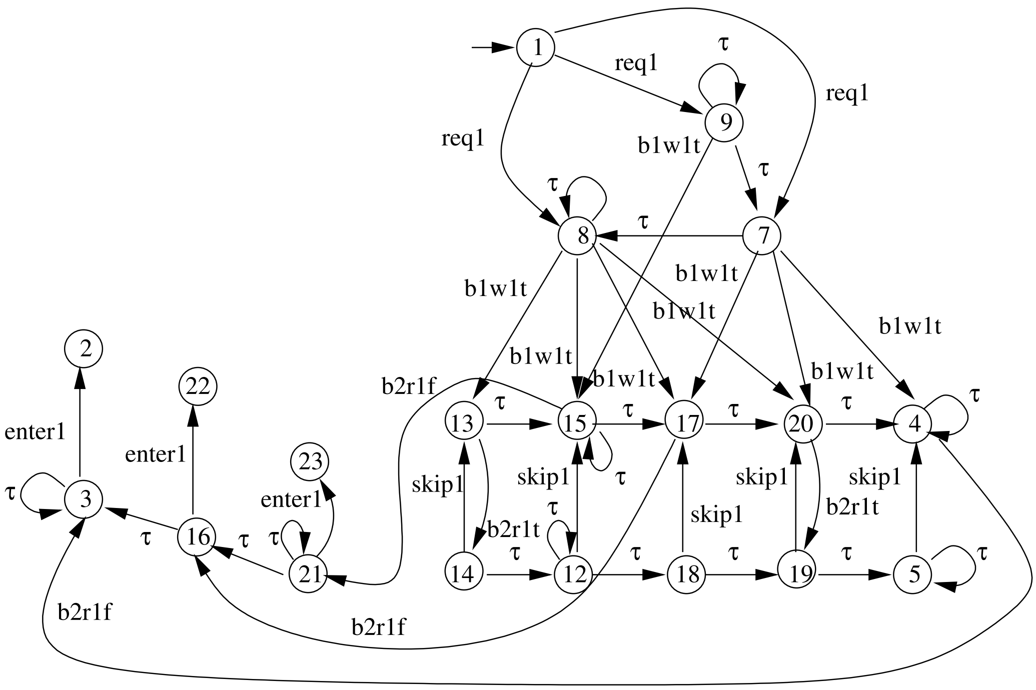


Figure 6: The lts $\hat{M}$ in the case of process $P_1$ of Lamport's algorithm

# 6 Discussion

This research begun with experimental work based on an intuitive idea given in a more precise form in propositions 21 and 22. The need to modify the D-bisimilarity [Elor94] to SD-bisimilarity and to assume biactiveness (all-activeness) arose in those experiments. The same reasons which prevent the usage of D-bisimilarity in our progress studies — the divergence loops need to be placed exactly at the right state in the minimized lts — prevented also the usage of cffd-equivalence [VaTi95] or any other trace-based divergence-preserving equivalence. They are simply too coarse for this kind of analysis.

The theory explaining the experimental results was developed afterwards. It was developed in a generalized form in order to cover also strong fairness and more than two parallel processes sharing the same memory. Unfortunately, the part of theory concerning progress checking using minimization after suitable label hidings can handle only biactive systems of two processes under weak pro-

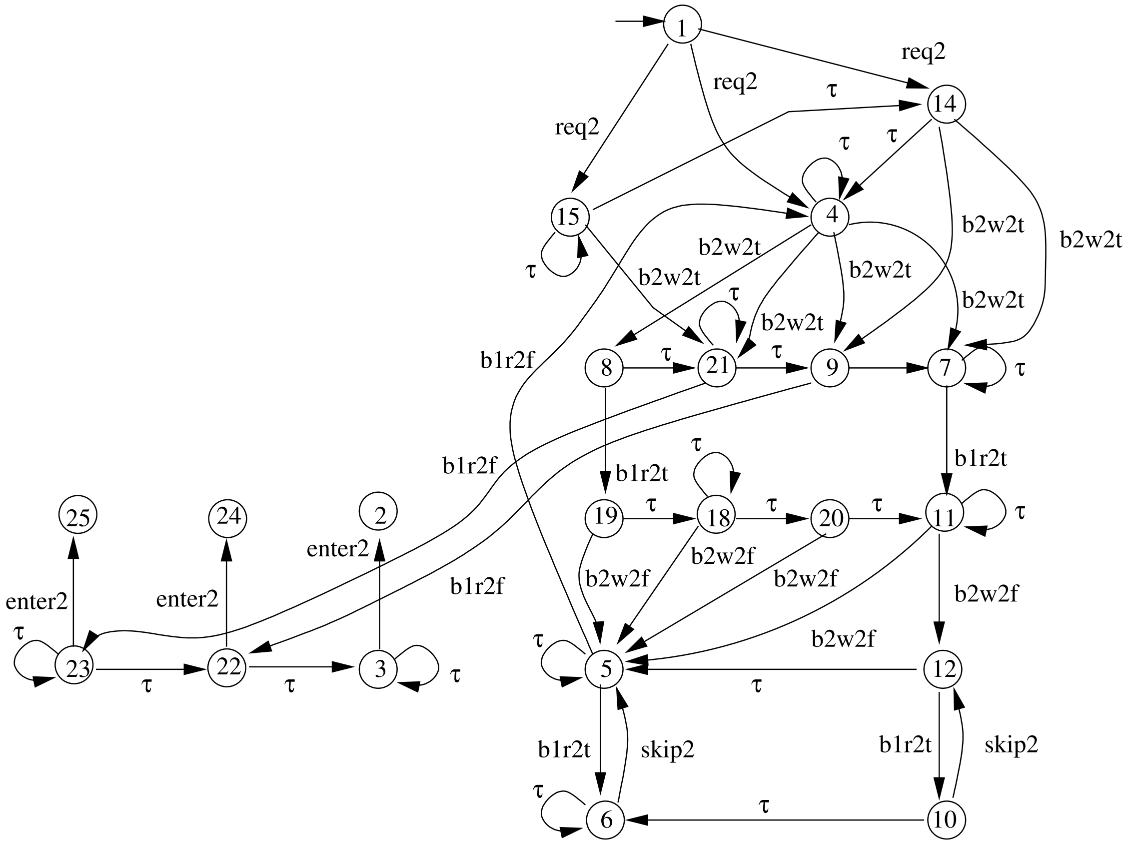


Figure 7: The lts $\hat{M}$ in the case of process $P_2$ of Lamport's algorithm

cess fairness assumption. An obstacle preventing our attempts of generalization has been the sentence "We assume further that the $Act(P_1)$-actions of $M'$ do not change the state of the *Memory*-component in $G$" in Proposition 18. If we could prove the proposition without this assumption, we could generalize our method to several concurrent processes.

We have not yet implemented the methods developed in Section 3, which would make it possible to check the progress properties under both strong and weak fairness assumptions. These methods could be generalized to be applicable for more general concurrent systems than our generic shared-memory system model (2) at page 4. These methods, where lts's are not minimized but are handled directly, have the advantage that they are formulated and available for arbitrary number of concurrent processes. They were found as byproducts of our research aiming actually at studying progress properties when lts's are condensed.

Our algorithms in Section 3 resemble the more general ones given for temporal logics model checking in [LiPn84]. A direct comparison is difficult because the problem formulation is so much more general in [LiPn84] and our algorithms cannot be directly derived from theirs as special cases. However, one can find, e.g., in [LiPn84, page 102] a recursive procedure to search for strongly fair non-progress executions which is very similar to our test function $Live(SSG)$ on the page 8. In [MaPn84, pages 25-26] one can find — for testing the liveness formula $q \rightarrow \Diamond\phi$ under weak fairness assumption — a condition equivalent with our our Condition 11 and a lemma equivalent with our Lemma 12. Our algorithms have been, however, developed independently of these earlier results.

As to the fairness assumptions, in our progress checking approach based on condensed systems, weak process fairness is used. When strong process fairness is assumed we have been unable to develop any method to deduce liveness properties by inspecting only the condensed, minimized system. If we could define a new version of strong fairness in a global-state-oriented form, e.g., "if an execution path passes infinite times a global state $s$ of $G$ every transition enabled at $s$ will eventually be taken", we could probably develop a compositional method for deducing progress properties. However — after having pursued this idea for a moment — we came to the conclusion that this interpretation is not permissible and not consistent with the mainstream fairness thinking. Later on we have found that this kind of fairness concept has been used in a recent Ph.D Thesis [Gian99] and in the TRACTA system analysis tool. It remains to be seen whether the new fairness concept suggested in [Gian99] will get general legitimacy.

We have aimed at a general method, verifying progress (Proposition 22) as well as non-progress (Proposition 21), from condensed lts's. Our aim has been to achieve visual verification. However, e.g., in Peterson's algorithm the visual verification presented in [VaSe96] is much more appealing. This has been achieved by using a coarser behavioural equivalence and by hiding more transitions than is possible in our method.

The way algorithms are modelled is essential in the method of [VaSe96]. There a modelling approach is used where, e.g., polling of variable values is abstracted away and thus no divergence loop caused by polling will appear at the states between $req_1$- and $enter_1$-transitions. If there would be a divergence in some of these states one would be in doubt whether it hampers progress or not. In [VaSe96] the fairness assumptions needed are not given explicitly. Obviously, some implicit fairness assumptions are necessary in that approach, too. Strong process fairness assumption would guarantee that all participating processes will execute transitions and proceed.

In our method polling of variable values is modelled explicitly. Polling of variable value $b2$ can be seen in the condensed global state graph of Petersons's algorithm (see Figure 5) as visible loops 12 $\stackrel{\mathrm{b2r1t}}{\longrightarrow}$ 11 $\stackrel{\mathrm{kr12}}{\longrightarrow}$ 13 $\stackrel{\mathrm{skip1}}{\longrightarrow}$ 12 and 5 $\stackrel{\mathrm{b2r1t}}{\longrightarrow}$ 7 $\stackrel{\mathrm{kr12}}{\longrightarrow}$ 6 $\stackrel{\mathrm{skip1}}{\longrightarrow}$ 5. In order to conclude that these loops are not preventing the progress of the process $P_1$ we needed to use the Proposition 18 (included in Proposition 22).

Obviously one could check all the progress properties we have handled in this paper using some model-checker based on temporal logic. However, applying very general methods, these are not easy-to-use software tools. They are mostly based on Kripke structures instead of labeled transition systems, have special restrictive input formats and model size limitations etc. Anyway, our goal has been to develop direct algorithms for checking progress properties of labeled transition systems, which could be used as an alternative, allowing handling of bigger systems than is possible with the established model checking tools.

# References


[Andr91] Andrews, G. R.. *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, Redwood City, California, 1991.

[BoBr87] Bolognesi, T. and Brinksma, E.: Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, Vol. 14, pp. 25 – 59, 1987.

[ClPS89] Cleaveland, R., Parrow, J., and Steffen, B.: The Concurrency Workbench. In *Automatic Verification Methods for Finite State Systems*, pp 11–23, Lecture Notes in Computer Science 407, Springer-Verlag, 1989.

[Gian99] Giannakopoulou,D.: Model Checking for Concurrent Software Architectures, Ph. D. Thesis at the Department of Computing, Imperial College of Science, Technology and Medicine, London, 1999.

[Elor94] Eloranta, J.: Minimal Transition Systems with Respect to Divergence Preserving Behavioural Equivalences. Ph.D. Thesis, University of Helsinki, Department of Computer Science, Report A-1994-1, Helsinki 1994, pp 1-162.

[Kuup2000] Kuuppelomäki, Päivi: Verifying liveness properties in some well-known shared memory mutual exclusion algorithms. University of Helsinki, Department of Computer Science, Internal report, (under preparation).

[Lamp86] Lamport, Leslie: The mutual exclusion problem, Part II, – statement and solutions, Journal of ACM, Vol. 33, No 2, 1986.

[LiPn84] Lichtenstein, O., Pnueli, A.: Checking That Finite State Concurrent Programs Satisfy Their Linear Specification, Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages, ACM Press, pp 97 – 107, 1984.

[MaPn84] Manna, Z., Pnueli,A.: Proving Precedence Properties: The Temporal Way, Technical Report CS84-04, The Weizmann Institute, Rehovot, Israel, February 1984.

[MaPn95] Manna, Z., Pnueli, A.: *Temporal Verification of Reactive Systems • Safety •*, Springer-Verlag, New York, 1995.

[Miln89] Milner, R.: *Communication and Concurrency*, Prentice-Hall, Great Britain, 1989.

[VaSe96] Valmari, A., Setala, M.: Visual Verification of Safety and Liveness, Proc. FME (Formal Methods Europe)'96, Lecture Notes of Computer Science, Vol. 1051, pp. 228-247, Springer Verlag, 1996

[VaTi95] Valmari, A., Tienari, M.: Compositional Failure-Based Semantic Models for Basic LOTOS. *Formal Aspects of Computing*, Vol. 7, pp. 440–468, 1995.

[Walk89] Walker,D.J.: Automated Analysis of Mutual Exclusion Algorithms using CCS. *Formal Aspects of Computing*, Vol. 1, pp. 273–292, 1989.