# Weak and branching bisimulation in Fctool

Amar Bouali

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

N° 1575

*Programme 2*
*Calcul Symbolique, Programmation*
*et Génie logiciel*

# WEAK AND BRANCHING BISIMULATION IN FCTOOL

**Amar BOUALI**

**Janvier 1992**

*RR-1575*

# Weak and Branching Bisimulation in FCTOOL
# Equivalence Observationnelle et Branching Bisimulation dans FCTOOL

Amar Bouali
ENSMP-CMA Sophia Antipolis
06565 Valbonne cedex
FRANCE
email:amar@cma.cma.fr

**Abstract**

R.J. van Glabbeek and W.P. Weijland introduced in [12] Branching Bisimulation, a new equivalence between transition systems: Branching Bisimulation refines the traditional Weak Bisimulation and has a lot of nice properties: simple algebraic characterization, complete term rewriting system, modal logic characterizations, preservation of action refinement. J.F. Groote and F. Vaandrager [4] presented an efficient algorithm for the Relational Coarsest Partition problem which can decide Branching Bisimulation on finite state labeled transition systems. In this paper we present this equivalence and an implementation inspired from the one given in [4]. From this implementation, we derive an new and efficient manner to decide Weak Bisimulation improving space consumption with respect to previous implementations. Our main work is the realization of a tool called FCTOOL gathering these new functionalities, and a *fc format* file interface exchange to let the Process Calculi Verification Tool AUTO [14] invoke FCTOOL.We also present a series of AUTO bench tests on typical examples.

**Résumé.**

R.J. van Glabbeek et W.P. Weijland ont introduit dans [12] la Branching Bisimulation, une nouvelle équivalence sur les systèmes de transitions: elle raffine l'équivalence observationnelle et possède de bonnes propriétés: caractérisation algébrique simple, axiomatisation complète, caractérisation par logiques modales, conservation par raffinement d'action. J.F. Groote and F. Vaandrager dans [4] ont présenté un algorithme efficace, basé sur une solution du problème du raffinement de partition, qui calcule la Branching Bisimulation sur des systèmes de transitions finis. Dans ce papier, on présente cette équivalence et une implémentation inspirée de celle donnée dans [4]. A partir de cette implémentation, on a mis en forme une nouvelle manière de calculer la Weak Bisimulation, qui manifestement améliore la complexité en espace comparée à la méthode traditionnelle. L'ensemble du travail se résume par la mise en forme d'un outil appelé FCTOOL, rassemblant l'algorithmique de ces nouvelles fonctionnalités, ainsi qu' une interface par fichier *fc format* avec l'outil de Vérification AUTO, pour permettre à ce dernier d'appeler les nouvelles procédures quand besoin est. Nous terminons par une série de sessions AUTO dans lesquelles nous traitons des exemples standards.

# Introduction

It is admitted that Branching Bisimulation has to be taken into account in the world of verification of distributed systems modeled by transitions systems. It lies between Strong and Weak (Observational) Bisimulation equivalences. It has a lot of nice properties, including an efficient algorithm for deciding it. In this paper, we present an implementation of this algorithm inspired directly from [4], from which we derive a new method to decide Weak Bisimulation avoiding the space problem due to the cost of the transitive closure computation of the transition relation. These functionalities are implemented in the so-called FCTOOL. To make AUTO capable to call these functions, we made an interface in between them, consisting in text files containing automata description in the so-called *fc format*. Our work is a practical example of use of such a format dedicated to interface verification tools based on transition system. AUTO produces automata from process calculus terms, FCTOOL computes equivalence classes of a bisimulation, and AUTO get back results for minimization, comparisons, further analysis...

Section 1 recalls Branching Bisimulation's formal definition on labeled transition systems and the Relational Coarsest Partition Problem algorithm which computes equivalence classes of bisimulation like equivalences, allowing minimization and equivalence decision.

Section 2 presents an implementation of this algorithm for the different instanciations for each kind of bisimulation. Indeed, we shall see that a general algorithm exists for all kind of bisimulation relativised by what we call a proper *instability notion*. Formal definitions and decision procedures are given for this notion. We derive then a new way to compute Weak Bisimulation without computing the whole transitive closure of the transition relation needed in this case.

Lasts sections are devoted to our implementations summed up in FCTOOL, to the *fc format* interface, and to the experiments where we collect some contrasting in between Branching Bisimulation and Weak Bisimulation: we found an example for which minimization by Branching Bisimulation and Weak Bisimulation don't yield the same result. We also evaluate efficiency of the implementations and the AUTO integration.

# 1 Algorithm for Bisimulations

We recall first some basic definitions about transition systems and bisimulations.

## 1.1 Bisimulations and Transition Systems

**Definition 1.1** Transition Systems
*A transition system is a structure* $A =< \mathcal{L}, \mathcal{S}, s_0, \longrightarrow >$, *where*

- $\mathcal{L}$ *is a set of labels, called actions,*
- $\mathcal{S}$ *is a finite set of states,*
- $s_0 \in \mathcal{S}$ *is the initial state,*
- $\longrightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ *is the transition relation.* $s \xrightarrow{a} s'$ *stands for* $(s, a, s') \in \longrightarrow$.

**Notation 1.1** *Let* $\{n\}$ *denotes the set of the* $(n+1)$ *first integers including* 0, *and* $\{n\}^*$ *the set of the* $n$ *first integers excluding* 0. *Throughout the paper, symbols* $\sim$, $\simeq$ *and* $\simeq_b$ *stand respectively for Strong Bisimulation, Weak Bisimulation and Branching Bisimulation equivalence. TS is a short hand for Transition System. We always refer to a TS* $A =< \mathcal{L}, \mathcal{S}, s_0, \longrightarrow >$ *and its components except when explicit notations are introduced.*

**Definition 1.2** Bisimulations

*Let* $\mathcal{A}_1 =< \mathcal{L}_1, \mathcal{S}_1, s_0^1, \longrightarrow_1 >$ *and* $\mathcal{A}_2 =< \mathcal{L}_2, \mathcal{S}_2, s_0^2, \longrightarrow_2 >$ *be two TS.* $\mathcal{A}_1 \Xi \mathcal{A}_2$, *where* $\Xi \in \{\sim, \simeq, \simeq_b\}$ *if there exists a binary symmetric relation* $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ *such that*

1. $s_0^1 \mathcal{R} s_0^2$,
2. $\forall s, s' : s \mathcal{R} s', \forall a \in \mathcal{L} \cup \{\tau\}$ *if* $s \xrightarrow{a} t$ *then*
   - $\Xi = \sim$:
     $\exists t' : s' \xrightarrow{a} t' \wedge t \mathcal{R} t'$.
   - $\Xi = \simeq_b$:
     *either* $a = \tau \wedge t \mathcal{R} s'$,
     *or* $\exists t', \exists p_0, p_1, \ldots, p_n, \exists q_0, q_1, \ldots, q_m$,
     $s' = p_0 \xrightarrow{\tau} p_1 \ldots \xrightarrow{\tau} p_n = t_1 \xrightarrow{a} t_2 = q_0 \xrightarrow{\tau} q_1 \ldots \xrightarrow{\tau} q_m = t'$ *s.t.*
     $t \mathcal{R} t' \wedge \forall i \in \{n\} \forall j \in \{m\}, s \mathcal{R} p_i \wedge t \mathcal{R} q_j$.
   - $\Xi = \simeq$:
     $\exists t' : s' \stackrel{a}{\Longrightarrow} t' \wedge t \mathcal{R} t'$, *where the relation* $\Longrightarrow$ *is defined as follow:*
     $\stackrel{\tau}{\Longrightarrow} = (\xrightarrow{\tau})^*$,
     $\forall a, a \neq \tau, \stackrel{a}{\Longrightarrow} = \stackrel{\tau}{\Longrightarrow} \xrightarrow{a} \stackrel{\tau}{\Longrightarrow}$.

**Definition 1.3** Quotient of $TS$

*Let* $\mathcal{A} =< \mathcal{L}, \mathcal{S}, s_0, \longrightarrow >$ *be a transition system and* $\equiv$ *an equivalence relation over the states. We write* $[s]$ *the equivalence class of the state* $s$. *We define the quotient of* $\mathcal{A}$ *by the relation* $\equiv$, *written* $\mathcal{A}_{\equiv}$, *as being the TS* $\mathcal{A}_{\equiv} =< \mathcal{L}_{\equiv}, \mathcal{S}_{\equiv}, s_0^{\equiv}, \longrightarrow_{\equiv} >$ *where* $\mathcal{S}_{\equiv} = \{[s] \mid s \in \mathcal{S}\}$, $\mathcal{L}_{\equiv} = \mathcal{L}$, $s_0^{\equiv} = [s_0]$, $\longrightarrow_{\equiv} = \{[s] \xrightarrow{a}_{\equiv} [s'] \mid s \xrightarrow{a} s', \forall a \in \mathcal{L}\}$.

## 1.2 Relational Coarsest Partition and Transition System

In this section we present the general algorithm for the Relational Coarsest Partition. This algorithm due to Kannellakis and Smolka [5] computes equivalence classes of bisimulation like equivalences over transition systems. We apply it for our various bisimulations using appropriate instability notion.

### 1.2.1 The General Algorithm

```
Π ← Π0
While Π not stable do
begin
      Find (B,B') unstable ;
      Π ← Ref_Π(B,B');
fin
```

Figure 1: The general algorithm

**Definition 1.4** Partitions

*The set* $\Pi = \{\mathcal{B}_i \subseteq \mathcal{S} \mid i \in \mathcal{I}\}$ *is a* **partition** *of* $\mathcal{S}$ *if and only if* $\bigcup_{i \in \mathcal{I}} \mathcal{B}_i = \mathcal{S}$ *and for* $i \neq j : \mathcal{B}_i \cap \mathcal{B}_j = \emptyset$.

*Elements of a partition are called* **blocks**. *If* $\Pi$ *and* $\Pi'$ *are two partitions of* $\mathcal{S}$, *then* $\Pi'$ **refines** $\Pi$ *if and only if* $\forall \mathcal{B}' \in \Pi', \exists \mathcal{B} \in \Pi$ *such that* $\mathcal{B}' \subseteq \mathcal{B}$.

In the rest of the paper, $\Pi$ stands for a partition of the set of states $S$. The algorithm maintains a partition $\Pi$ initialized to $\Pi_0$, the one element partition, and repeats a refinement step which consists in finding an *unstable* pair of blocks (definition is given later) and refining the current partition with respect to it, until a stable partition is obtained. This algorithm, shown in figure 1, is used to compute bisimulation equivalences with some related instability notion. Given a set of states of some $TS$ and an instability notion corresponding to some bisimulation $\approx$, the algorithm computes a partition $\Pi = \{\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_n\}$ such that each $\mathcal{B}_i$ corresponds exactly to an equivalence class of $\approx$. In the next section we formalize the instability notion for the bisimulations we are interested in. In the case of Strong Bisimulation Weak Bisimulation it is taken from [5], and for Branching Bisimulation from [4].

### 1.2.2 Instability Notions

**Definition 1.5** Instability

*Given an action $a$ in $\mathcal{L}$, a pair of blocks $(\mathcal{B}, \mathcal{B}')$ of $\Pi$ is* **a-unstable** *iff in the setting of*

- $\sim$
  $\emptyset \neq Pos_a(\mathcal{B}, \mathcal{B}') \neq \mathcal{B}$ *where* $Pos_a(\mathcal{B}, \mathcal{B}') = \{s \in \mathcal{B} \mid \exists s' \in \mathcal{B}', s \xrightarrow{a} s'\}$.
- $\simeq$
  $\emptyset \neq Pos_a(\mathcal{B}, \mathcal{B}') \neq \mathcal{B}$ *where* $Pos_a(\mathcal{B}, \mathcal{B}') = \{s \in \mathcal{B} \mid \exists s' \in \mathcal{B}', s \stackrel{a}{\Longrightarrow} s'\}$.
- $\simeq_b$
  $(\mathcal{B} \neq \mathcal{B}'$ *or* $a \neq \tau)$ *and* $\emptyset \neq Pos_a(\mathcal{B}, \mathcal{B}') \neq \mathcal{B}$ *where*

$$\begin{aligned} Pos_a(\mathcal{B}, \mathcal{B}') \;=\; & \{s \in \mathcal{B} \mid \exists s_1, s_2, \ldots, s_n, s' \\ & s_0 = s, \forall i \in \{n\}^\star, s_i \in \mathcal{B} \wedge s_{i-1} \xrightarrow{\tau} s_i, \\ & s_n \xrightarrow{a} s' \wedge s' \in \mathcal{B}'\}. \end{aligned}$$

*A pair of blocks is said* **unstable** *in any setting previously described, iff there exists an action $a$ for which these blocks are a-unstable, otherwise it is said* **stable**. *More generaly, a block $\mathcal{B}$ of $\Pi$ is said stable iff for all block $\mathcal{B}'$ in $\Pi$, the pair $(\mathcal{B}, \mathcal{B}')$ is stable. The partition $\Pi$ is stable iff each of its block is stable. Otherwise, it is unstable.*

**Definition 1.6** Refinement

*If the pair of blocks $(\mathcal{B}, \mathcal{B}')$ of $\Pi$ is a-unstable, then a* **refinement** $Ref^a_\Pi(\mathcal{B}, \mathcal{B}')$ *of $\Pi$ is obtained by replacing $\mathcal{B}$ with the blocks $Pos_a(\mathcal{B}, \mathcal{B}')$ and $\mathcal{B} - Pos_a(\mathcal{B}, \mathcal{B}')$.*

### 1.2.3 The case of Branching Bisimulation instability notion

The instability notion we have described for Branching Bisimulation is taken from [4], and there it is shown that it can be checked efficiently considering some properties on states. Actually, two kinds of states inside a block are distinguished: so, for a block $\mathcal{B}$, $bottom(\mathcal{B})$ is the subset of states of $\mathcal{B}$ which have no $\tau$-transition ending in $\mathcal{B}$, that is formally,

$$bottom(\mathcal{B}) = \{s \in \mathcal{B} \mid s \xrightarrow{\tau} s' \Longrightarrow s' \notin \mathcal{B}\}$$

Other states in $\mathcal{B}$ are potentially equivalent to those in $bottom(\mathcal{B})$. In [4], the authors argue that the analysis of instability can be essentially done in $bottom(\mathcal{B})$. In fact a pair of blocks $(\mathcal{B}, \mathcal{B}')$ of $\Pi$ is a-unstable iff

- either, $(bottom_a(\mathcal{B}), \mathcal{B}')$ is a-unstable, where formally
  $bottom_a(\mathcal{B}) = \{r \in bottom(\mathcal{B}) \mid \exists s, r \xrightarrow{a} s\}$,
- or, $Pos_a(bottom_a(\mathcal{B}), \mathcal{B}') = \emptyset \wedge \exists r \in \{\mathcal{B} - bottom_a(\mathcal{B})\}, \exists s \in \mathcal{B}'$ s.t. $r \xrightarrow{a} s$.

### 1.2.4 The case of Weak Bisimulation instability notion

The instability notion we have previously presented for this case is related to the usual way to compute Weak Bisimulation, namely:

1. a preprocessing step to compute the so-called "double arrow" by a transitive closure of the transition relation with respect to the set of regular expressions $\{\tau^\star, \tau^\star a \tau^\star\}$,
2. the usual computation of the strong bisimulation using the previous extended transition relation.

We propose another way of deciding Weak Bisimulation, reducing the space consuming essentially due to the transitive closure. We take here advantage of the efficiency of Branching Bisimulation data structures and marking techniques to stay in a same order of time complexity than before. So, we first extend the transition relation partially, adding only the transitions with respect to $\{\tau^+\}$, building the following "double arrow":

1. $s \stackrel{\tau}{\Longrightarrow} s' \iff \exists n > 0, \exists s_0, s_1, \ldots, s_n, s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_n = s'$,
2. $s \stackrel{a}{\Longrightarrow} s' \iff \exists s \xrightarrow{a} s'$

For no confusion with our new double arrow, we write $\Longrightarrow_\tau$ as the traditional extended transition relation for Weak Bisimulation. Then we consider the following instability notion, which is three folded:

1. - A pair of blocks $(\mathcal{B}, \mathcal{B}')$, $\mathcal{B} \neq \mathcal{B}'$ of $\Pi$ is **$\tau$-unstable** iff
     $\emptyset \neq Pos_\tau(\mathcal{B}, \mathcal{B}') \neq \mathcal{B}$ where $Pos_\tau(\mathcal{B}, \mathcal{B}') = \{s \in \mathcal{B} \mid \exists s' \in \mathcal{B}', s \stackrel{\tau}{\Longrightarrow} s'\}$
   - Let $\Pi^\tau_{\mathcal{B}'}$ be the partition obtained from $\Pi$ replacing all $\tau$-unstable blocks $\mathcal{B}$ w.r.t. $\mathcal{B}'$ by $Pos_\tau(\mathcal{B}, \mathcal{B}')$ and $\mathcal{B} - Pos_\tau(\mathcal{B}, \mathcal{B}')$.
   - Let $\Psi_{\mathcal{B}'} = \{\mathcal{B} \in \Pi^\tau_{\mathcal{B}'} \mid \mathcal{B} \stackrel{\tau}{\Longrightarrow} \mathcal{B}'\} \cup \{\mathcal{B}'\}$.
2. For $a \neq \tau$ let $\Psi^a_{\mathcal{B}'} = \{\mathcal{B} \in \Pi^\tau_{\mathcal{B}'} \mid \emptyset \neq Pos_a(\mathcal{B}, \mathcal{B}') \neq \mathcal{B}\}$ where
   $Pos_a(\mathcal{B}, \mathcal{B}') = \{s \in \mathcal{B} \mid \exists s' \in \Psi_{\mathcal{B}'}, s \stackrel{a}{\Longrightarrow} s'\}$.
3. Then a pair of blocks $(\mathcal{B}, \mathcal{B}')$ is **a-unstable** iff
   $\mathcal{B} \in \Psi^a_{\mathcal{B}'}$ or $\exists \mathcal{B}_a \in \Psi^a_{\mathcal{B}'}$ such that $(\mathcal{B}, \mathcal{B}_a)$ is $\tau$-unstable.

We call it *weak instability notion.* One can ask about the ability of this technique of deciding Weak Bisimulation. To give an answer we state the following proposition.

**Proposition 1.1** *The relational coarsest partition general algorithm with the weak instability notion decides Weak Bisimulation.*

**Proof:** It suffices to prove that when considering a block $\mathcal{B}'$ in a partition $\Pi$, the refinement step w.r.t this block is the same than the one in the traditional way to compute Weak Bisimulation. Precisely, we show that in our three steps, we find all the block $\mathcal{B}'$ that are $a$-unstable in the setting of Weak Bisimulation, for some action $a$ . We distinguish two cases:

1. if $a = \tau$, then our first step find all $\tau$-unstable blocks w.r.t. $\mathcal{B}'$,
2. if $a \neq \tau$, then
   - the second step finds the following blocks:
     $\Psi_1 = \{\mathcal{B} \mid \exists s \in \mathcal{B}, s \xrightarrow{a} \mathcal{B}'\} \cup \{\mathcal{B} \mid \exists s \in \mathcal{B}, \exists \mathcal{B}_\tau \in \Psi_{\mathcal{B}'}, s \xrightarrow{a} \mathcal{B}\}$.
     That is, it finds the blocks $\mathcal{B}$ containing at least one state having an $a$-transition ending either in $\mathcal{B}'$ or in a block belonging to $\Psi_{\mathcal{B}'}$. This done rewinding the transition relation w.r.t. $E = \{\xrightarrow{a} \cup \xrightarrow{a} \stackrel{\tau}{\Longrightarrow}\}$
   - the third step finds the blocks:
     $\Psi_2 = \{\mathcal{B} \mid \exists s \in \mathcal{B}, s \stackrel{\tau}{\Longrightarrow} \mathcal{B}_1 \wedge \mathcal{B}_1 \in \Psi_1\}$, that is, those $\mathcal{B}$ containing at least one state from which there is a $\tau$-path ending in a block belonging to $\Psi_1$. They are obtained rewinding the transition relation from $\Psi_1$ w.r.t. $\stackrel{\tau}{\Longrightarrow}$.

Thus the rewinding from $\mathcal{B}'$ to $\Psi_2$ is done w.r.t. $\stackrel{\tau}{\Longrightarrow} E$ which is exactly equivalent to the usual one step rewinding w.r.t. $\tau^* a \tau^*$. □

In the sequel, we present an implementation of the general algorithm with the previous instability notions and the integration of the ability to decide Branching Bisimulation by the verification tool AUTO.

# 2 Implementations

We introduce the abstract data type inspired from [4], to represent transition systems and partitions to achieve the required efficiency. Some notations will be justified by the selected programming language, namely C++.

## 2.1 Adopted Structures

A $TS$ is implicitly represented by a structured partition containing all its states and transitions. The figure 2 gives an idea of the design of the data structures we adopt.

A partition is a set of blocks. **A block** $\mathcal{B}$ is seen as a structure consisting of

- `BottomList`: the state list $bottom(\mathcal{B})$,
- `NonBottomList`: the state list $\mathcal{B} \setminus bottom(\mathcal{B})$,
- `TransitionList`: the list of transitions having a target state belonging to $\mathcal{B}$.

**A state** is a structure containing

- `StateRef`: an integer which is its representative,
- `TauList`: its list of $\tau$-transition,
- `BlockRef`: a pointer reference to the block it belongs to.

**A transition** is also a structure consisting of

- `Source`: its source state number.
- `Action`: an integer reference to an action of $\mathcal{L}$,
- `Target`: its target state number.

We add specific fields to each structure for making double-linked lists. For example, a partition is seen as a list of block structures.

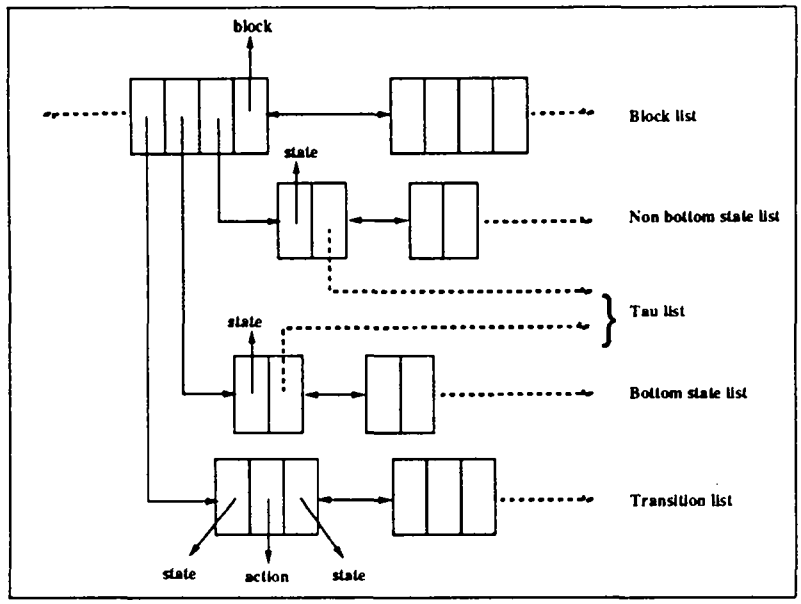



Figure 2: The data structures

## 2.2 Basic objects and functions

Let T be a transition structure. We make use of the functions `source(T)`, `label(T)`, `target(T)` which give respectively the source state number, the labeling action number, the target state number of the transition T.
Let L be a double linked list of elements $l_1, l_2, \ldots, l_n$. We make use of following functions:

- `size(L)`=$n$, `first(L)`=$l_1$, `last(L)`=$l_n$ return respectively the size, the first element, the last element of L.
- `next(`$l_i$`)`, `prev(`$l_i$`)`: return respectively the next element, the previous element of $l_i$ in L if respectively $l_i \neq$ last(L), $l_i \neq$ first(L) and return NIL otherwise.
- `addfirst(`$l$`, L)`, `addlast(`$l$`, L)` add $l$ respectively in head, in tail of L, if it doesn't still appear in.
- `empty(L)`, `match(`$l$`, L)`: return true if respectively `size(L)` $> 0$, if $l \in$ L, false otherwise.
- `delete(`$l$`, L)`: delete the element $l$ from L.

If B is a block, then `sort(B)` is the function defined as the following set of actions

$$sort(\mathcal{B}) = \{a \in \mathcal{L} \mid \exists s \in B \text{ such that } s \xrightarrow{a} s' \wedge (a = \tau \Longrightarrow s' \notin \mathcal{B})\}$$

## 2.3 The Procedures

We describe the set of procedures we have implemented to compute the considered bisimulations. Some of them are common to all kind of bisimulation. The essential difference appears in the instability notion computation. A preprocessing step is required for Branching Bisimulation and Weak Bisimulation.

### 2.3.1 Branching and Weak Bisimulation Preprocessing

We begin by a deletion of $\tau$-cycles, merging all the states contained in such a cycle. We use here an algorithm from [1] solving the problem of finding all strongly connected components in a graph. This allows us to sort topologically the states in the graph restricted to transitions labeled by $\tau$. Then we sort the other transitions (`TransitionList`) with respect to their labeling action, and the order is kept after any modification made to lists of transitions.

In the case of Weak Bisimulation we add a preprocessing step in which we compute our kind of extended transition relation. We recall that unlike the traditional way, we don't compute the usual "double arrow" extension, that is the transitive closure of the transition relation with respect to the regular expressions of actions $\{\tau^*, \tau^* a \tau^*, \forall a \in \mathcal{L}\}$. We just compute the transitive closure w.r.t. the expression $\{\tau^+\}$. We show in a later section how we use marking techniques in conjunction with this new kind of extended relation to compute Weak Bisimulation.

### 2.3.2 The Main Loop

We start with an initial partition containing a unique block gathering together all the $TS$'s states and transitions. Throughout the algorithm, we maintain two lists of blocks. One is called `StableBlock`: it contains the blocks with respect to which each block of the current partition is stable. The other one, called `UnstableBlock` collects the blocks with respect to which stability conditions have to be checked for each block of the partition. So initially the former is empty and the latter contains the unique block of the initial partition.

The main loop of the algorithm is the refinement of the partition with respect to a found unstable pair of blocks, until we get a stable partition, that is:

```
procedure MainLoop() {
   while not empty(UnstableBlock) do {
      B — first(UnstableBlock);
      delete(B, UnstableBlock);
      addfirst(B, StableBlock);
      CheckUnstableBlock(B);
   }
}
```

More precisely, `MainLoop` takes a block from `UnstableBlock` and transfers it temporally in `StableBlock`. Then, the procedure `CheckUnstableBlock` takes each block of the current partition and checks a stability condition with respect to the first taken block and refines the current partition if necessary. We write it as:

```
procedure CheckUnstableBlock(B) {
   ∀a ∈ sort(B) and while first(StableBlock)=B do {
      BlockList — Unstable(a, B);
      Refine(B, BlockList);
   }
}
```

`CheckUnstableBlock` is a loop over the sort of `B` while it doesn't appear as an unstable block (w.r.t. itself), in which case we stop this loop refining the partition splitting the block `B`, and restart the main loop with the new obtained partition. Here, an instability

checking function (detailed in later sections) is called to find potentially unstable blocks with respect to B and return a list of such blocks. This list is given to the procedure Refine written as:

```
procedure Refine(B, BlockList) {
   B' ← first(BlockList);
   do {
      Split(B');
      B' ← next(B');
   } while B' ≠ NIL and first(StableBlock)=B;
}
```

BlockList contains all the blocks which are presumed unstable (w.r.t. B). The procedure Split is devoted to determine whether they are or not effectively unstable. If a block is unstable, the procedure splits it in two sub-blocks, with one containing the marked states and appropriate transitions. Thus, it refines the current partition and transfers the new blocks in the unstable block list. One can remark that the procedure stops when B is found unstable with respect to itself. In what follows, we distinguish the spliting functions which is the same for Strong Bisimulation and Weak Bisimulation, Split1, but different for Branching Bisimulation, Split2, having to split a block in two steps: one for the bottom states and one for the non bottom ones.

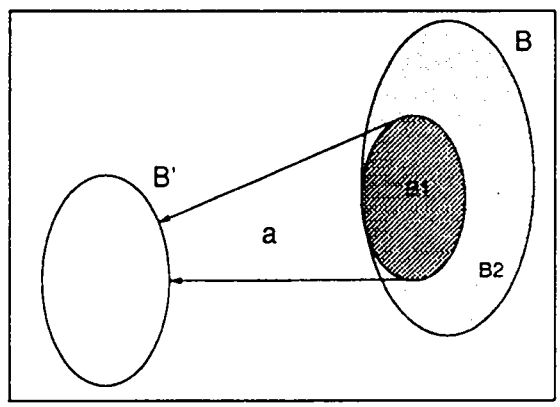
```
procedure Split1(B) {
   B1 ← B;
   B2 ← SplitBottom(B1);
   if not empty(B2.BottomList) then {
      SplitTransition(B1,B2);
      addfirst(B2,UnstableBlock);
   }
}


procedure Split2(B) {
   B1 ← B;
   B2 ← SplitBottom(B1);
   if not empty(B2.BottomList) then {
      SplitNonBottom(B1,B2);
      SplitTransition(B1,B2);
      addfirst(B2,UnstableBlock);
   }
}


procedure SplitBottom(B1) {
   Let B2 be a block;
   S ← first(B1.BottomList);
   do {
      if S marked then {
      if match(B1, StableBlock)=true then {
         delete(B1, StableBlock);
         addfirst(B1, UnstableBlock);
```

```
        }
        delete(S, B1.BottomList);
        addfirst(S, B2.BottomList);
        }
        S ← next(S);
    } while S ≠ NIL;
    return B2;
}


procedure SplitTransition(B1,B2) {
    T ← first(B1.TransitionList);
    do {
        if target(T) marked then {
        delete(T,B1.TransitionList);
        addfirst(T,B2.TransitionList);
        }
        T ← next(T);
    } while T ≠ NIL;
}


procedure SplitNonBottom(B1,B2) {
    S ← first(B1.NonBottomList);
    do {
        if S ⟹^τ S' and S' marked then {
        delete(S, B1.BottomList);
        addfirst(S, B2.BottomList);
        }
        S ← next(S);
    } while S ≠ NIL;
}
```

Here we present the methods used to determine the possibly unstable blocks with respect to a given block for each case we are interested in.

### 2.3.3 Strong Bisimulation Instability Notion

The following picture shows the marking step used in this case. We want to check the stability of B′ with respect to B.

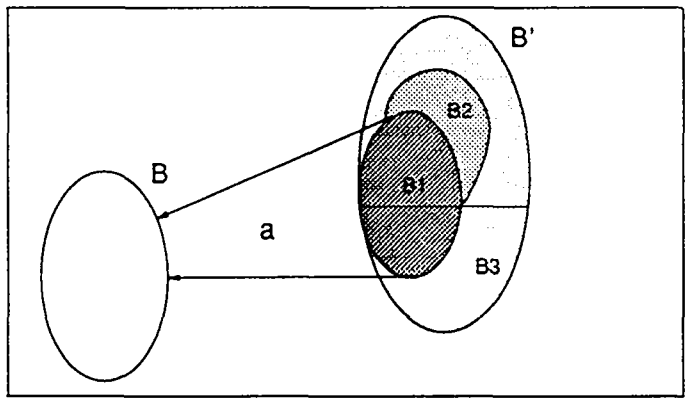


For this, we mark in B′ the state having an outgoing transition reaching a state in B labeled by a certain action, say a. If B′ has at least one of such a state, then it is added in the list of possibly unstable blocks.

```
procedure Unstable(a, B) {
   T — first(B.TransitionList);
   BlockList ← NIL;
   do {
      if label(T)=a then {
      mark source(T);
      addfirst(source(T).BlockRef, BlockList);
      }
      T ← next(T);
   } while T ≠ NIL;
}
```

### 2.3.4 Branching Bisimulation Instability Notion

In this case, the following picture helps us to see that it consists in marking the states having an outgoing transition to B labeled by an action a. In a second step, we mark also the state which can perform a $\tau$ path reaching a marked state. To do this well and efficiently, we take advantage of having a topological sorting on such states. This comes from the deletion of $\tau$-cycles, as already mentioned.



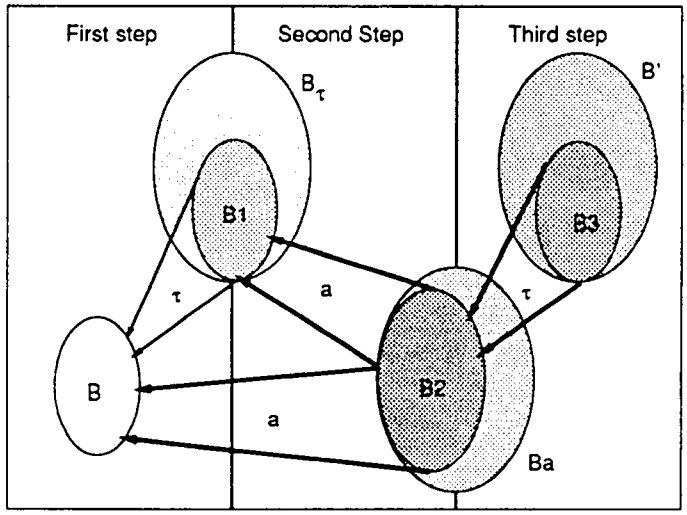

The picture symbolises the procedure showing B1 as the first obtained subset of marked stated in the analysed block, and B2 as the subset of states obtained from the extension of this first step to the non bottom states. The cross line in a block stands for the separation of bottom states from the others, put downward by convention.

```
procedure Unstable(a, B) {
   T — first(B.TransitionList);
   BlockList — NIL;
   do {
      if label(T)=a then {
      mark source(T);
      addfirst(source(T).BlockRef, BlockList);
      }
      T — next(T);
   } while T ≠ NIL;
```

```
    ExtendeMarking(a, B);
    return BlockList;
}


procedure ExtendeMarking(a, B) {
    S ← fisrt(B.NonBottomStates);
    do {
        if S not marked and S --τ--> S' and S marked then {
        mark S;
        }
        S ← next(S);
    } while S ≠ NIL;
}
```

### 2.3.5 Weak Instability Notion

We consider here our kind of extented transition relation. The following picture gives an idea of the different marking steps we proceed in this case. Actually there are three steps to check the weak instability notion.




The first step looks for a first list of blocks in $\Pi \setminus \{B\}$ which have at least one state having an outgoing $\tau$-transition ending in B. We then split the blocks of this list separating marked states from the others (in the picture, $B_\tau$ is splited into B1 and $B_\tau$-B1), gathering in a list called `FirstBlockList` the blocks containing marked states. This step is performed by `FirstSplit`.

The second step makes a second list, `SecondBlockList`, of blocks from $\Pi$ (such as $B_a$) which contains at least one state having an outgoing transition labeled by a and ending either in a block B1 from `FirstBlockList`, or in B.

In a third step, we add in `SecondBlockList` the blocks which contain at least a state performing a $\tau$-transition ending in a block contained in `SecondBlockList` (like B′ in the figure). The procedure `TauUnstable` finds such blocks. This list is finally returned by the procedure `Unstable`.

```
procedure Unstable(a, B) {
```

```
   FirstBlockList ← NIL;
   FirstSplit(B, FirstBlockList);
   addfirst(B, FirstBlockList);
   C ← first(FirstBlockList);
   SecondBlockList ← NIL;
   do {
      T ← first(C.TransitionList);
      do {
         if label(T)=a then {
            mark source(T);
            addfirst(source(T).BlockRef, SecondBlockList);
         }
         T ← next(T);
      } while T ≠ NIL;
      C ← next(C);
   } while C ≠ NIL;
   D ← first(SecondBlockList);
   do {
      TauUnstable(D, SecondBlockList);
      D ← next(D);
   } while D ≠ NIL;
   return SecondBlockList;
}

procedure FirstSplit(B, BlockList) {
   TauUnstable(B, BlockList);
   C ← first(BlockList);
   do {
      SplitBottom(C);
      C ← next(C);
   } while C ≠ NIL;
   return BlockList;
}

procedure TauUnstable(B, BlockList) {
   T ← first(B.TransitionList);
   do {
      if label(T)=τ then {
         mark source(T);
         addfirst(source(T).BlockRef, BlockList);
      }
      T ← next(T);
   } while T ≠ NIL;
   return BlockList;
}
```

The effectiveness of the instability property on those blocks containing at least one marked state is then decided as follow: if such a block contains at least one unmarked state, then it is unstable. This comes directly from the instability conditions we have expressed. Before the presentation of the integration of these functions in AUTO, we state some results about the complexity of the algorithms.

### 2.4 Complexity

In [4], it is proved that Branching Bisimulation can be decided in $\mathcal{O}(mn)$ time complexity, where $m$ is the number of transitions and $n$ the number of states; $\mathcal{O}(m)$ to check stability and to proceed a one step refinement when necessary; (upto) $n$ times this step to obtain a stable partition. The worst case is when we obtain the partition containing $n$ blocks each containing a single state. The space complexity is in $\mathcal{O}(m)$.
Within FCTOOL, this complexity is also the same for Strong Bisimulation because we use the algorithm of [5] but it can be done more efficiently with Paige and Tarjan algorithm [8] ($\mathcal{O}(m \log n)$ when the number of labels is considered as a constant); for Branching Bisimulation we inherit the complexity of the implementation exposed in [4]. Our new method to compute Weak Bisimulation is done with the same machinery as Branching Bisimulation except when we check the instability notion: there, we use the same "rewinding" treatments of the transition relation as in Branching Bisimulation. In fact, our three steps procedure yields three times the complexity we have in the case of Branching Bisimulation. This factor doesn't change the time complexity. The space complexity is $\mathcal{O}(m')$, where $m'$ is the number of transition of the TS obtained after the $\tau^+$ transitive closure.
One could think that we loose some complexity in time in comparison with the standard way to compute Weak Bisimulation (full transitive closure + Paige and Tarjan); in fact, it is hard to compare the two methods and to state which algorithm is more efficient, since in our case we deal with a smaller TS than in the original version. The fact is that each time Branching Bisimulation is more efficient than the usual algorithm for Weak Bisimulation (e.g in ALDÉBARAN), so will be our improved algorithm. We will discuss about this again in the report of our experiments in section 4.

## 3 Integration in AUTO

Instead of using its own bisimulation procedures, now AUTO can call the various functions of FCTOOL via an *fc format* file interface: the quotient of a transition system with respect to Strong Bisimulation, Branching Bisimulation, and Weak Bisimulation. We have two ways to deal with Weak Bisimulation:

1. by our procedure with restricted extended transition relation and marking techniques,
2. first compute a quotient w.r.t. Branching Bisimulation, then use Weak Bisimulation on it.

In a later section, we discuss the time benefits of these methods. Before, we describe the way of calling the tool inside AUTO.

### 3.1 The *fc format* interface

The system AUTO [14] is written in LE-LISP. Our implementation being in C++ we had to interface these tools with text files containing automata descriptions to be or having been processed. We choose the *fc format*, taking advantage of its recent development. This format is the result of a joint work of INRIA/IMAG Verification Tool team. Briefly, it is a simple syntax for describing graph like objects in text files supporting several semantical aspects verification tools may need (structural, behavioural, model-checking, logical, "private hooks"). It is devoted to interface tools based on automata like objects processing. It allows a tool to give data to proceed to another tool and to get back

resulting data. The format consists in a two part description of an automaton: an *info* part, and a *skeleton* part. In the latter, the structure of the automaton itself is given, by lists of states and (labeled) transitions, all represented by integers. These integers refers to expression table entries making up the former part. This first part gathers the names and (possibly) properties of the elements composing the automaton. Thus, our work is also an experiment showing some of facilities of such a format. Up to now, AUTO, ALDÉBARAN, CWB [3] are able to deal with this format. A recent improvement has been done on it: the new version allow one to define static networks of processes, describing its components and their synchronization and communication activities via synchronization vectors [2].

### 3.2 The Calls in AUTO

AUTO has a simple command language for calling its functionalities allowing variable environment interface for storing and reusing results. Until now, functions `mini()` and `obs()` in AUTO are respectively Strong Bisimulation and Weak Bisimulation. We add `branching()` with the same syntax as preceding functions for Branching Bisimulation. These reduction functions when applied to a term of a process calculus, say CCS, are called with respect to congruent properties of the operators present in the term. Now, these functions can call in background ours (except Branching Bisimulation which always calls FCTOOL) which appear to be a real speed-up for AUTO, as we show it in next sections. This call is effectively done depending on an integer value characterizing a minimal number of state above which FCTOOL is called. Under this limit, AUTO still calls its own functions. This size is stored in a variable called `hoggar-floor`: it can be set by the user. Larger presentation can be found in [13].

## 4 Examples

Several authors have reported recently on the practical efficiency and capacity of finite systems analysis tools [6,4,10]. AUTO doesn't have the capacity of building and handling very large structures. Our work increases its power, allowing to it to treat larger examples than before. To show it, we begin with a session with the famous Milner's Scheduler problem which was the one chosen in previously cited papers. Automata graphical representation are made by AUTOGRAPH [11]. We close the examples section with the analysis of a mutual exclusion algorithm due to Peterson. A reference to the Peterson's solution of the problem can be found in [9].

There are two general ways for AUTO to treat a CCS term. One is to compile it as a product of automata (each automaton coming from a sequential component of the network denoted by the term). This product results in the transition system related to the global system. Then minimization can be applied on it.

In the other way, AUTO takes advantage of congruent properties of the considered process algebra operators (in the case of CCS, the parallel operator is congruent for the bisimulation equivalences we are intersted in). In fact, two manners are provided: the first is to compute directly the minimization on the term structurally, that is, the parallel operator being binary, subproducts are computed on each pair of components; then the result is minimized and composed with an other minimized subproduct, and so on until the minimized global TS is obtained. This method allows AUTO to avoid storing intermediate TS which can be much larger than the reduced ones; the other is to minimize the sequential components (the leaves of the term tree), and to compute the automaton in a one pass residual algorithm (Precise explainations can be found in [13]). Experiments with this

latter method lead AUTO to compute in about 30 seconds of CPU time the minimal weak observational equivalent TS of the scheduler with 128 cyclers.
The experiments we present in this paper are made applying the first general way, to make correct comparisons with results exposed in previously cited papers. But this is not meaningful, because this does not correspond to the AUTO's powerful way of manipulating terms for verification as the previous shortly cited result shows it (complete presentation is exposed in [13]).
All the experiments are made in a SUN server 4/390 with 24MB of memory.

### 4.1 Milner's Scheduler

The reader shall refer to [7] for a complete specification. We show in figure 3 an AUTOGRAPH specification from which we get a CCS term for AUTO to analysis. As shown by our main reference paper, Branching Bisimulation and Weak Bisimulation yield the same results, the former being more efficient than the latter.
The scheduler CCS term, compiled into an automaton by AUTO represents the global system of the scheduler leting all the actions visible.

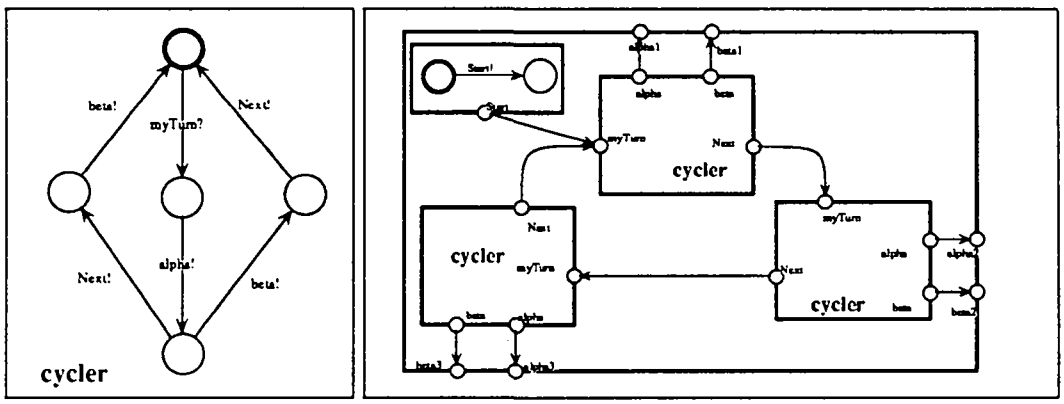



Figure 3: An AUTOGRAPH specification of a scheduler with three cyclers

**FCTOOL: comparing Weak Bisimulation and Branching Bisimulation**

| k | states | trans. | Branching | Weak. |
|---|---|---|---|---|
| 4 | 97 | 241 | < 0.1s | < 0.1s |
| 5 | 241 | 721 | 0.2s | 0.3s |
| 6 | 577 | 2017 | 0.7s | 1.0s |
| 7 | 1345 | 5377 | 2.4s | 3.1s |
| 8 | 3073 | 13825 | 6.7s | 9.0s |
| 9 | 6913 | 34561 | 19.5s | 25.7s |
| 10 | 15361 | 84481 | 53.6s | 72.8s |

Figure 4: FCTOOL performances

As the interface between AUTO and FCTOOL is made by *fc format* text files, the power of the latter system is not totally inherited by the former. Thus, comparisons with Branching

Tool (BB) with respect to Weak Bisimulation and Branching Bisimulation is not totally satisfactory. For this reason, we made comparisons of the two equivalences within FCTOOL separately from AUTO, that is with exactly the same conditions for both. Results are collected in table 4. The times given in this table include input and output time of the *fc format* interface, which takes about 50% of the total time. The numbers of state and transitions are those of the global system's $TS$ before reduction (obtained in AUTO using `tta()`).
Figure 4 shows that in FCTOOL, the time taken to compute Weak Bisimulation in FCTOOL is about 25% more than the one taken to compute Branching Bisimulation. So our way to compute Weak Bisimulation is an efficient version.

FCTOOL: **a speed-up for** AUTO

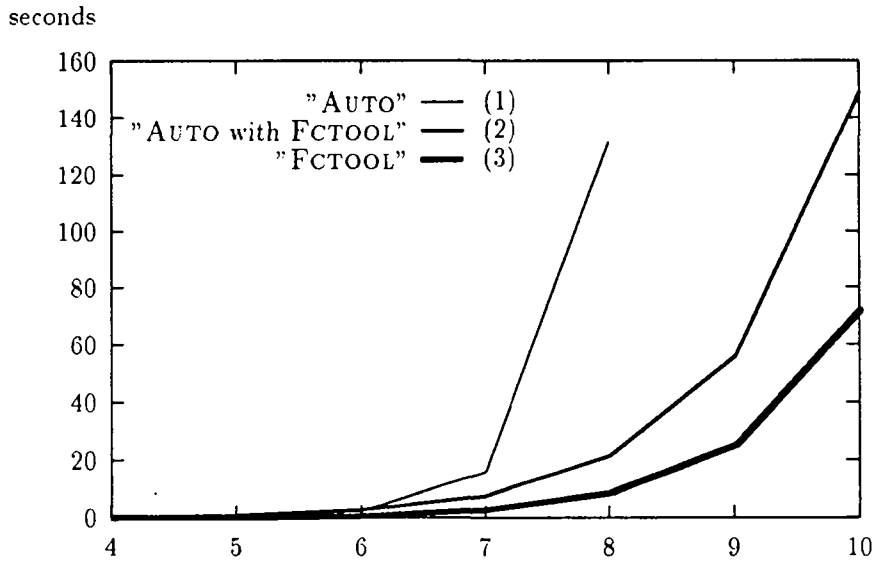



Figure 5: AUTO New Performances

Figure 5 contains the results we obtain when we apply on it the Weak Bisimulation reduction procedure, calling AUTO's function (curve 1), and calling FCTOOL (curve 2) via *fc format* files. The curve 3 represents the time taken by FCTOOL within AUTO. All time results include read/write operations relating to the *fc format*, the reduction computation and the quotient TS construction. One can see the gain in AUTO when calling FCTOOL instead of its own functions: in this case, FCTOOL adds efficiency in time and space allowing AUTO to treat larger examples than before. Times considered in the figure contains the time of the algorithm and the one spent in the reading writing *fc format* files. In this example a reasonable value for `hoggar-floor` appears to be around 1000.

## 4.2 Peterson's Mutual Exclusion Algorithm

We took mutual exclusion algorithms to analyse Branching Bisimulation. We present here the more interesting one, namely the Peterson's solution, being an example where Branching Bisimulation and Weak Bisimulation do not coincide. Briefly, several processes (2 in our case) behave concurrently to accede to a critical section (CS for short). A protocol is designed to make the processes enter and exit the CS fairly. It consists in testing a boolean value and a turn variable shared by all the processes. The invariant is

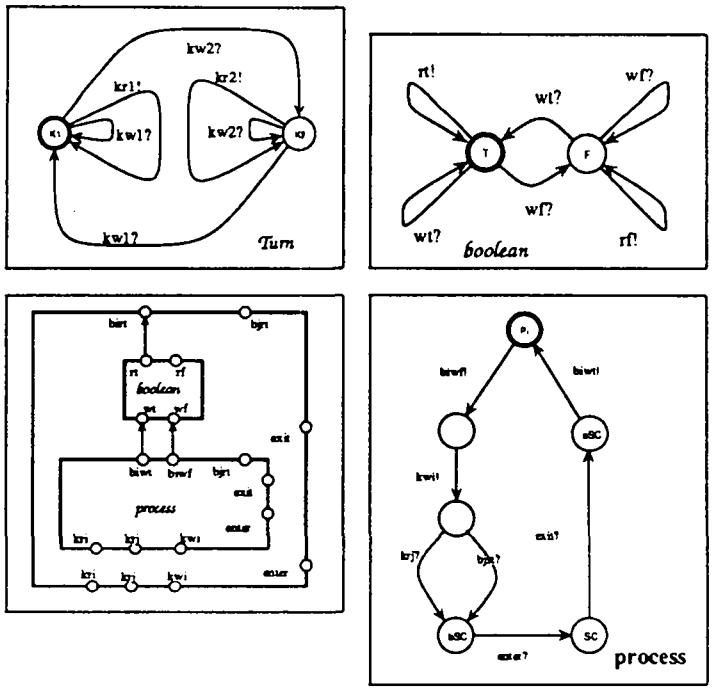



Figure 6: Mutual exclusion: Peterson's solution

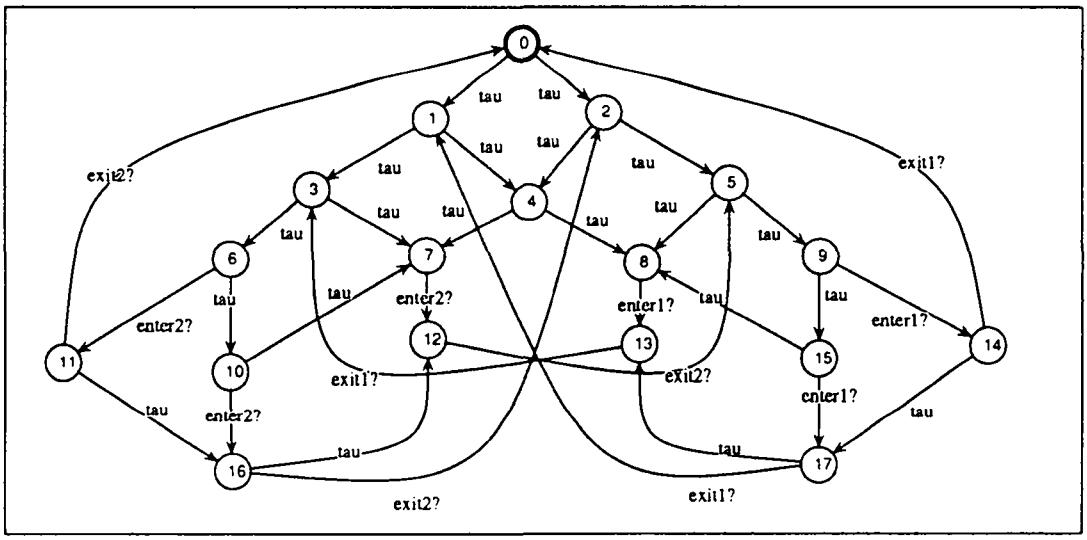



Figure 7: Reduction by Branching Bisimulation

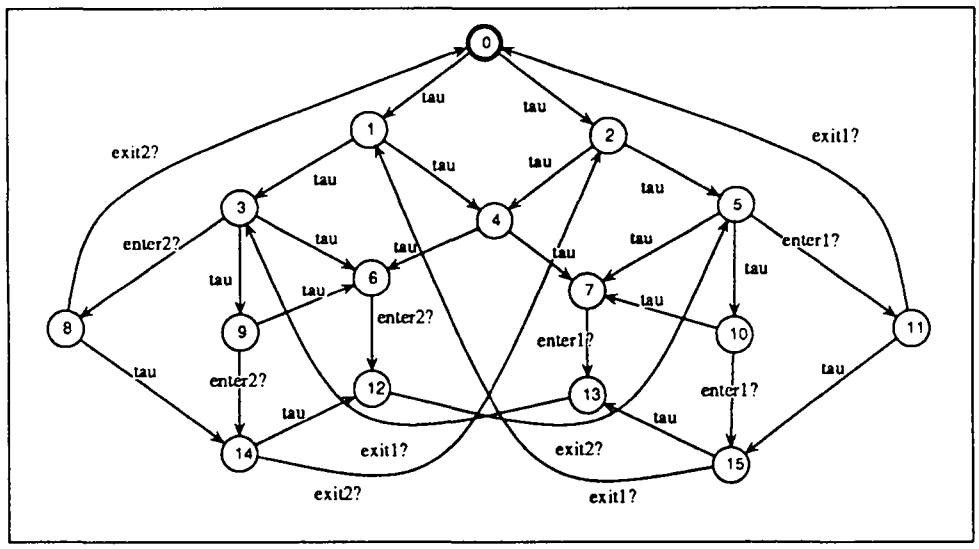



Figure 8: Reduction by Weak Bisimulation

that a precise boolean condition is always verified by at most one process, to make sure that only one process can be in the CS.

Figure 6 is a graphical specification of the solution made with AUTOGRAPH, from wich we obtain a CCS term. The AUTO session given in annex.

Figures 7 and 8 show the reduced transition system by respectively Branching Bisimulation ($Q_1$) and Weak Bisimulation($Q_2$).

It is easy to verify that states 3 and 6 in $Q_1$ are observationally equivalent and are represented by the state 3 in $Q_2$. Similarly, states 5 and 9 in $Q_1$ are merged in state 5 in $Q_2$. All we can say about state 3 in $Q_1$ is valid for state 5 in $Q_1$. One can remark that states 3 and 6 are not branching bisimilar. It is due to the fact that from state 3 there are two different ways to perform a $\tau$ action that lead to different future behaviours. One is to get into state 6 and the other to state 7. The former allows the process 2 to accede the CS leading to two possible futures: one is the case where the two processes are conflicting to the CS access (going back to state 0, via state 11), and the other is the case where only the process 1 will be able to accede to the CS (going to the state 5 via state 12). Regarding the state 6, there is only one $\tau$ behaviour leading to the two possible futures. So the choice of those two futures is made at different moment from state 3 and 6. For the state 3, it is done during the performance of the $\tau$ action, while for the state 6 it is done after the $\tau$ action. This situation is the same for the states 5 and 9.

# 5 Conclusion

The aim of our work was to include in AUTO the ability to compute Branching Bisimulation. For efficiency reasons, we did a C++ implementation of the algorithm described in [4]. The data structures and marking techniques for the partition refinement introduced in this paper helped us to develop a new way to compute Weak Bisimulation. In the same time, we took advantage of the development of the so-called *fc format* to realise a concrete experiment with it, interfacing our tool with AUTO.

The conclusion we can formulate is the speed-up we get for AUTO when it calls FCTOOL

to compute quotient of transition systems w.r.t bisimulation equivalences. Also, we have shown that our way to compute Weak Bisimulation, which is less space consuming than the standard way to compute it, is performed about the same time Branching Bisimulation. During example sessions, we found a case for which Branching Bisimulation and Weak Bisimulation don't yield the same result. So Branching Bisimulation can't replace totally Weak Bisimulation; it can just help to have "less" things to do when we want to compute Weak Bisimulation. That is, when one wants to reduce a transition system with respect to observational equivalence, it can be done exploiting Branching Bisimulation with which one can obtain a first quotient on which Weak Bisimulation is applied.
Our CCS specification of the Peterson's mutual exclusion algorithm is a kind of counter example of the feelings expressed in [4] about the fact that Branching Bisimulation and Weak Bisimulation coincide for a large class of processes. The problem that still arise is to formally characterize a class of processes for which Branching Bisimulation and Weak Bisimulation coincide.

## Acknowledgments

The author would like to thank Robert de Simone for the general design of the implementations of FCTOOL and integration of Branching Bisimulation in AUTO, Eric Madelaine for reading and commenting on successive drafts of the paper, and Didier Vergamini for his comments about AUTO and on implementation details.

## References


[1] A. Aho, J. Hopcroft, and J. Ullman. *Design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] André Arnold. Sémantique des processus communicants. *RAIRO Informatique Théorique*, 15(2):103–139, 1981.

[3] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Workshop On Computer-Aided Verification*, DIMACS, 1990.

[4] J.F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. *ICALP '90*, 1990.

[5] P.C. Kanellakis and S.A. Smolka. Ccs expressions, finite state processes, and three problems of equivalence. In *ACM Symposium on Principles of Distributed Computing*, pages 228–240, 1983.

[6] H.P. Korver. *The Current State of Bisimulation Tools.* Report P9101, University of Amsterdam, 1991.

[7] Robin Milner. *Communication and Concurrency. International Series in Computer Science*, Prentice Hall, 1989.

[8] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM*, 16(6), 1987.

[9] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[10] Huajun Qin. *Efficient Verification of determinate Processes.* Technical Report, Dep. of Comp. Sc., SUNY, Stony Brook, 1991.

[11] Gresse V. Roy. *Autograph: un outil de visualisation de processus parallèles et communiquants.* PhD thesis, Université de Nice, 1990.

[12] R.J. Van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). *Information processing '89 (G.X. Ritter, ed.) Elsevier Science*, 613–618, 1984.

[13] D. Vergamini. AUTO/MAUTO *user manual V2-3.* Technical Report (to appear), INRIA, 1991.

[14] D. Vergamini. *Vérification de réseau d'automates finis par equivalence observationnelle: le système AUTO.* PhD thesis, Université de Nice, 1987.

# A The Peterson AUTO Session

```
$ auto
                                    AUTO
                            Version 2.2 (08 Mar 1991)

@ load "peterson";
@ set ccs=true;
ccs : Bool

@ load "Turn";
@ parse Turn = let rec {K1 = kr1!:K1 + kw2?:K2 + kw1?:K1
meije0>  and K2 = kr2!:K2 + kw1?:K1 + kw2?:K2} in K1 ;
Turn : Process of meije0

@ file Turn.ec loaded.

@ load "boolean";
@ parse boolean = let rec {T = wt?:T + rt!:T + wf?:F
meije0>  and F = wf?:F + rf!:F + wt?:T} in T ;
boolean : Process of meije0

@ file boolean.ec loaded.

@ load "process";
@ parse process = let rec {Pi = biwf!:st_1
meije0>  and st_1 = kwi!:st_0
meije0>  and st_0 = krj?:bSC + bjrt?:bSC
meije0>  and bSC = enter?:SC
meije0>  and SC = exit?:eSC
meije0>  and eSC = biwt!:Pi} in Pi ;
process : Process of meije0

@ file process.ec loaded.

@ load "bool_and_process";
@ parse bool_and_process = (boolean[birt/rt,biwt/wt,biwf/wf]
// process)\rf\biwt\biwf ;
bool_and_process : Process of meije0

@ file bool_and_process.ec loaded.

@ load "mutex12net";
@ parse mutex =
(bool_and_process[bjrf/birf,bjrt/birt,birf/bjrf,birt/bjrt,cjrf/cirf,
cjrt/cirt,cirf/cjrf,cirt/cjrt,exit2/exit,enter2/enter]
// (bool_and_process[kwj/kwi,krj/kri,kwi/kwj,kri/krj,exit1/exit,
enter1/enter]
// Turn[kwj/kw1,krj/kr1,kwi/kw2,kri/kr2]))
\kwj\krj\kwi\kri\bjrf\bjrt\birf\birt\cjrf\cjrt\cirf\cirt;
```

```
mutex : Process of meije0

@ file mutex12net.ec loaded.

@ set peterson=tta mutex;
peterson : Automaton

@ file peterson.ec loaded.

@ display peterson short;
size = 32 states, 54 transitions, 5 actions.

@ set r_B=hoggar-branching peterson;
r_B : Automaton

@ display r_B short;
size = 18 states, 32 transitions, 5 actions.

@ set r_W=obs peterson;
r_W : Automaton

@ display r_W short;
size = 16 states, 30 transitions, 5 actions.

@
```

# B *fc format* files examples

The following files are the *fc format* exchange files in between AUTO and FCTOOL in the case of the previous AUTO session.

The global system

```
% automaton produced by auto %
% unit action "tau" %
automaton "peterson"
states 32
initial 0
atoms 15
0:"tau"
1:"enter1?"
2:"enter2?"
3:"exit1?"
4:"exit2?"
5:"Pi"
6:"T"
7:"K1"
8:"st_1"
9:"F"
10:"st_0"
11:"K2"
12:"bSC"
13:"SC"
14:"eSC"

monomials 0

state 0  trans 1
0:a0-> 2 1 2
state 1  trans 1
0:a0-> 2 3 4
state 2  trans 1
0:a0-> 2 4 5
state 3  trans 1
0:a0-> 2 6 7
state 4  trans 1
0:a0-> 2 7 8
state 5  trans 1
0:a0-> 2 8 9
state 6  trans 2
0:a0-> 11
1:a2-> 10
state 7  trans 1
0:a0-> 12
state 8  trans 1
0:a0-> 13
state 9  trans 2
0:a0-> 14
1:a1-> 15
state 10  trans 2
0:a0-> 17
1:a4-> 16
state 11  trans 2
0:a0-> 18
1:a2-> 17
state 12  trans 1
0:a0-> 18
state 13  trans 1
0:a0-> 19
state 14  trans 2
0:a0-> 19
1:a1-> 20
state 15  trans 2
0:a0-> 20
1:a3-> 21
state 16  trans 1
0:a0-> 2 22 23
state 17  trans 2
0:a0-> 24
1:a4-> 23
state 18  trans 1
0:a2-> 24
state 19  trans 1
0:a1-> 25
state 20  trans 2
0:a0-> 25
1:a3-> 26
state 21  trans 1
0:a0-> 2 0 26
state 22  trans 1
0:a0-> 2 27 28
state 23  trans 1
0:a0-> 2 28 29
state 24  trans 1
0:a4-> 29
state 25  trans 1
0:a3-> 30
state 26  trans 1
0:a0-> 2 1 30
state 27  trans 1
0:a0-> 2 3 31
state 28  trans 1
0:a0-> 2 5 31
state 29  trans 1
0:a0-> 5
state 30  trans 1
0:a0-> 3
state 31  trans 1
0:a0-> 2 7 8
end
```

The Branching Bisimulation reduced system

```
% automaton produced by fctool %
% unit action "tau" %
automaton "noname"
states 18
initial 0
atoms 15
0:"tau"
1:"enter1?"
2:"enter2?"
3:"exit1?"
4:"exit2?"
5:"Pi"
6:"T"
7:"K1"
8:"st_1"
9:"F"
10:"st_0"
11:"K2"
12:"bSC"
13:"SC"
14:"eSC"
monomials 0

state 0 trans 2
0: a0 -> 11
1: a0 -> 9
state 1 trans 2
0: a4 -> 0
1: a0 -> 15
state 2 trans 2
0: a0 -> 6
1: a0 -> 14
state 3 trans 2
0: a0 -> 12
1: a3 -> 0
state 4 trans 2
0: a0 -> 8
1: a0 -> 17
state 5 trans 1
0: a3 -> 4
state 6 trans 1
0: a1 -> 5
state 7 trans 1
0: a4 -> 2
state 8 trans 1
0: a2 -> 7
state 9 trans 2
0: a0 -> 10
1: a0 -> 2
state 10 trans 2
0: a0 -> 6
1: a0 -> 8
state 11 trans 2
0: a0 -> 10
1: a0 -> 4
state 12 trans 2
0: a0 -> 5
1: a3 -> 11
state 13 trans 2
0: a1 -> 12
1: a0 -> 6
state 14 trans 2
0: a1 -> 3
1: a0 -> 13
state 15 trans 2
0: a4 -> 9
1: a0 -> 7
state 16 trans 2
0: a0 -> 8
1: a2 -> 15
state 17 trans 2
0: a0 -> 16
1: a2 -> 1
end
```

The Weak Bisimulation reduced system

```
% automaton produced by fctool %
% unit action "tau" %
automaton "noname"
states 16
initial 0
atoms 15
0:"tau"
1:"enter1?"
2:"enter2?"
3:"exit1?"
4:"exit2?"
5:"Pi"
6:"T"
7:"K1"
8:"st_1"
9:"F"
10:"st_0"
11:"K2"
12:"bSC"
13:"SC"
14:"eSC"
monomials 0

state 0 trans 2
0: a0 -> 7
1: a0 -> 12
state 1 trans 3
0: a0 -> 10
1: a2 -> 4
2: a0 -> 8
state 2 trans 3
0: a0 -> 15
1: a1 -> 3
2: a0 -> 13
state 3 trans 2
0: a0 -> 14
1: a3 -> 0
state 4 trans 2
0: a4 -> 0
1: a0 -> 9
state 5 trans 1
0: a3 -> 1
state 6 trans 1
0: a4 -> 2
state 7 trans 2
0: a0 -> 11
1: a0 -> 2
state 8 trans 1
0: a2 -> 6
state 9 trans 2
0: a4 -> 7
1: a0 -> 6
state 10 trans 2
0: a2 -> 9
1: a0 -> 8
state 11 trans 2
0: a0 -> 13
1: a0 -> 8
state 12 trans 2
0: a0 -> 11
1: a0 -> 1
state 13 trans 1
0: a1 -> 5
state 14 trans 2
0: a3 -> 12
1: a0 -> 5
state 15 trans 2
0: a1 -> 14
1: a0 -> 13
end
```

ISSN 0249-6399