TAPAs: a Tool for the Analysis of Process Algebras

Francesco Calzolai, Rocco De Nicola, Michele Loreti, and Francesco Tiezzi

Dipartimento di Sistemi e Informatica Università degli Studi di Firenze
{calzolai,denicola,loreti,tiezzi}@dsi.unifi.it

Abstract. Process Algebras are a set of formalisms for modelling systems that permit mathematical reasoning with respect to a set of desired properties. TAPAs is a tool that can be used to support the use of process algebras to specify and analyze concurrent systems. Systems are described as process algebras terms that are then mapped to Labelled Transition Systems (LTSs). Properties are verified either by checking equivalence of concrete and abstract systems descriptions, or by model checking temporal formulae over the obtained LTS. A key feature of TAPAs is that a consistent double representation as term and as graph of each system is maintained. TAPAs does not aim at guaranteeing high performances, but is thought of as a support for teaching process algebras.

1 Introduction

Process Algebras are a set of mathematically rigorous languages with well defined semantics that allow for descriptions and verification of properties of concurrent communicating systems. They can be seen as mathematical models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. The agents may be real-word objects (even people), or they may be artefacts, embodied perhaps in computer hardware or software systems.

Process Algebras provide a number of constructs for system descriptions and are equipped with an operational semantics that describes systems evolution. Moreover they often come with observational mechanisms that permit identifying (through behavioral equivalences) those systems that cannot be taken apart by external observations. In some cases they came also equipped with sets of axioms, that capture the relevant identifications.

There has been huge amount of research work on process algebras carried out in the last 25 years started from the introduction of the theory of the process algebras CCS [18, 19], CSP [7] and ACP [5]. In spite of the many conceptual similarities they have been developed starting from quite different viewpoints and have given rise to different approaches: CCS relies on an observational bisimulation-based theory starting from an operational viewpoint. CSP was motivated as a theoretical version of a practical language for concurrency and is still based on operational view which, however, is interpreted w.r.t. a more abstract theory based on decorated traces. ACP started from a completely different viewpoint where concurrent systems are seen, according to a purely mathematical algebraic view, as the solutions of systems of equations (axioms) over the signature of the algebra considered; operational semantics and bisimulation (in this case a different notion of branching bisimulation is considered) are seen as just

one of the possible models over which the algebra can be defined and the axioms can be applied. At first, the different algebras have been developed completely separately. Slowly, however, the strict relationships have been understood and appreciated, nevertheless in university courses they have been taught separately. Thus we have seen many books on CCS [19], CSP [15, 21, 22], ACP [3, 11], Lotos [6] but not a book just on Process Algebras aiming at showing the underlying vision of the general approach. We feel that it is time to aim at teaching the general theory of Process Algebras and seeing the different languages as specific instances of the general approach.

The main ingredients of a specific process algebra are:

1. A minimal set of well thought operators aiming at capturing the relevant aspect of systems behavior and the way systems are composed.
2. A transition system associated to the algebra via structural *operational semantics* to describe the evolution of all systems that can be built from the operators.
3. An equivalence notion that permits abstracting from irrelevant details of systems descriptions.

Often process algebras come also equipped with:

4. Abstract structures that are compositionally associated to terms to provide *denotational semantics*.
5. A set of laws (axioms) that characterize behavioural equivalences to obtain a so called *algebraic semantics*.

Verification of concurrent system within the process algebraic approach is performed either by resorting to behavioural equivalences for proving conformance of processes to specifications that are expressed within the notation of the same algebra or by checking that processes enjoy properties described by temporal logics formulae [16, 8].

In the former case two descriptions of a given system, one very detailed and close to the actual concurrent implementation, the other more abstract describing the abstract trees of relevant actions the systems has to perform are provided and tested for equivalence.

In the latter case, concurrent systems are specified as terms of a process description language while properties are specified as temporal logic formulae. Labelled transition systems are associated to terms via a set of structural operational semantics rules and model checking is used to determine whether the transition systems associated to terms enjoy the property specified by the temporal formulae.

In both approaches Labelled Transition Systems (LTS) play a crucial role. They consist of a set of states, a set of transition labels and a transition relation. States correspond to the configurations systems can reach. Labels describe the actions systems can perform to interact with the environment. Transition relations describe systems evolutions as determined by the execution of specific actions. Temporal logic formulae are a mix of logical operators and modal operators. The former are the usual boolean operators, while the latter are those that permit reasoning about systems evolution in time and to deal with the dynamic aspects of LTS.

LTS are also the central ingredient of TAPAs the software tool that we have implemented to support teaching of process algebras, and whose component permit minimizing and animating LTS, testing their equivalence and model checking satisfaction of temporal formulae.

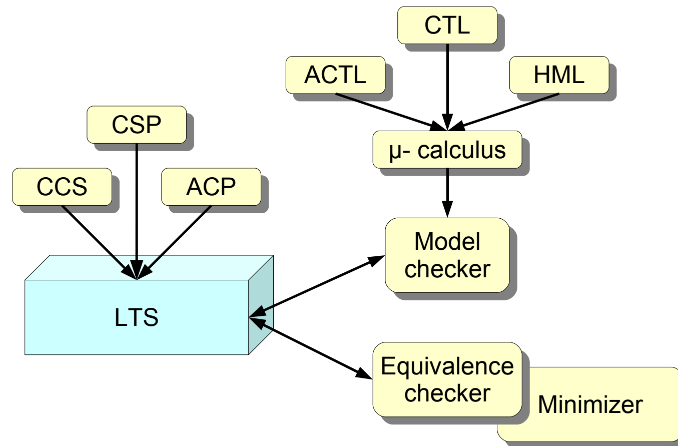

Fig. 1. TAPAs Architecture

By relying on a sophisticated graphical user interface TAPAs permits:

1. Understanding the meaning of the different process algebras operators by showing how they can be used to compose terms and the changes induced on the composed transition systems.
2. Appreciating the close correspondence between terms and processes by consistently updating terms when the graphical representation of labelled transition systems is changed and redrawing process graphs when terms are modified.
3. Evaluating the different behavioral equivalences by having them on a single platform and checking the different equivalences by simply pushing different buttons.
4. Studying model checking via a user friendly tool that in case of failures provides appropriate counterexamples that help debugging the specification.

The rest of the paper is organized as follows. In Section 2, we provide an overview of the TAPAs frontend and how, even using small examples, this can be used for specifying behaviours of concurrent systems. In Section 3, we describe the components that can be used for verifying system behaviour. In Section 4, we deal with a more elaborate case study. The final section contain a few concluding remarks.

2 Textual and graphical representation of processes

TAPAs[1] (Tool for the Analysis of Process Algebras [1]) is a graphical tool, developed in JAVA, which aims at simplifying the specification and the analysis of concurrent systems described by means of Process Algebras. This tool has been used for supporting

[1] TAPAs is a free software; it can be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation.

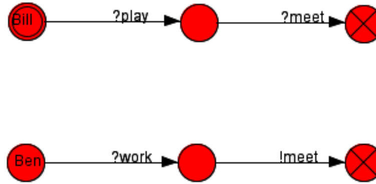

Fig. 2. Processes Bill and Ben

teaching of Theory of Concurrency in a course of the Computer Science curriculum at University of Firenze. TAPAs architecture is outlined in Figure 1. It consists of four components: an editor, a runtime environment, a model checker and an equivalence checker. TAPAs editor permits specifying concurrent systems as terms of a process algebra, with graphical and textual notation at the same time. The runtime-environment permits generating the Labeling Transition System corresponding to a given specification. Finally, model-checker and equivalence checker can be used for analyzing system behaviours. The former permits verifying whether a specification satisfies a logic formula of modal μ−calculus [16], the latter permits verifying if two implementations of the same system are equivalent or not, and to minimize Labeling Transition Systems with large number of states (by means of its subcomponent called minimizer).

In TAPAs concurrent systems are described by means of *processes*, which are nondeterministic descriptions of system behaviours, and *process systems*, which are obtained by process compositions. Notably, processes can be defined in terms of other processes or process systems. Processes and process systems are composed by using the operators of a given process algebra. For instance, in the case of CCS, a process system can be obtained by parallel composition, relabelling and restriction of processes.

TAPAs editor permits defining processes and process systems by using both graphical and textual representations. A process is graphically represented by a graph whose edges are labeled with the actions it can perform. The same process can be represented (textually) by a term of a specific process algebra. A user can always change the process representation: TAPAs will guarantee the synchronization between the graph and the corresponding term. We would like to remark that, TAPAs does not rely on a single process algebra that has to be used for the system specification. Indeed, even if currently only CCS has been considered, thanks to the modular implementation of TAPAs, other process algebras can be easily added.

Figure 2 shows two TAPAs processes that are the graphical representations of the following CCS terms:

$$Bill = ?play.\, ?meet.\, \mathtt{nil} \qquad\qquad Ben = ?work.\, !meet.\, \mathtt{nil}$$

The process *Bill* can perform an input on channel *play* and continue with an input on channel *meet*, while the process *Ben* can perform first an input on *work* and then an output on *meet*. It is worth noting that processes are named CCS terms, namely they represent declarations of the form $K = T$, where K is a constant (i.e. an identifier, as *Bill* or *Ben*, written in the initial state of the graph), and T is a standard CCS term.

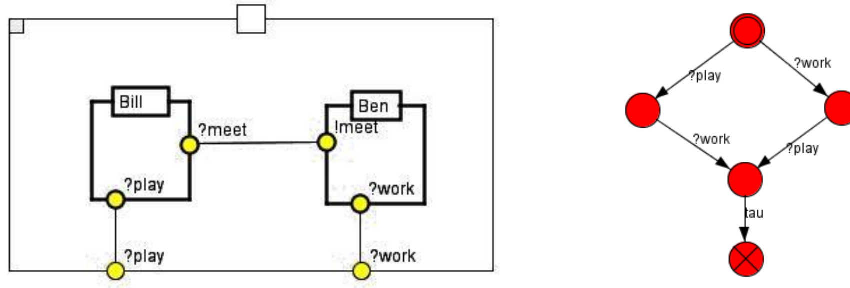

Fig. 3. Bill and Ben process system and LTS

Process systems, like processes, are represented both graphically and textually. In the first case, a system is represented by a box containing a set of elements. In the left side of Figure 3 contains the process system where the processes *Bill* and *Ben* are composed in parallel. To guarantee the synchonization between *Bill* and *Ben*, channel *meet* is restricted. This is represented graphically with a line connecting the ports ?`meet` and `!meet` on the internal boxes. These processes interact with the environment by using channels *play* and *work* that are not restricted. Indeed, internal ports ?`play` and ?`work` are connected with the corresponding ports on the external box. The textual representation of the process system is the following:

$$\begin{array}{l}\texttt{let}\\ \quad \mathit{Bill}\ = ?\mathit{play}.\,?\mathit{meet}.\,\texttt{nil};\\ \quad \mathit{Ben}\ = ?\mathit{work}.\,!\mathit{meet}.\,\texttt{nil};\\ \texttt{in}\ (\mathit{Bill}\ |\ \mathit{Ben})\backslash\{\mathit{meet}\};\end{array}$$

In the right side of Figure 3 is presented the Labeled Transition System corresponding to the above process system generated by the runtime component. To help the user to analyze the generated graph, TAPAs provides some visualization algorithms for drawing LTSs. New algorithms for drawing graphs can be easely plugged into TAPAs.

2.1 Textual specification of terms

The TAPAs runtime environment takes as input a textual specification, written in some process algebras, and generates the corresponding LTS, which can be used by the other components for model and equivalence checking. Currently, the only process algebras that can be used to specificate concurrent systems with TAPAs is CCS (Calculus of Communicating System [19]), one of the most known and used process calculi. However, additional modules for other process algebras can be easily developed and integrated in TAPAs.

In this section, we present the syntax of the CCS terms accepted by the runtime environment, that can be used for specifying systems textually. As we will see in Section 2.2, the set of CCS terms that can be graphically generated is a subset of the lan-

Left production		Right production	
T ::=	(CCS terms)	`let` D `in` S	(TAPAs CCS terms)
`nil`	(nil)		
$\mid$ K	(process name)	D ::=	(declarations)
$\mid$ $\alpha.T$	(action prefix)	$K = T;$	(term decl.)
$\mid$ $T + T$	(choice)	$\mid$ $D\ D$	(decl. list)
$\mid$ $T \mid T$	(parallel comp.)		
$\mid$ $T[f]$	(relabelling)	S ::=	(process systems)
$\mid$ $T \backslash C$	(restriction)	K	(call)
		$\mid$ $S \mid S$	(parallel comp.)
f ::=	(relabelling functions)	$\mid$ $S[f]$	(relabelling)
c/c	(new chan. / old chan.)	$\mid$ $S \backslash C$	(restriction)
$\mid$ $f\ ,\ f$	(relabelling list)		

Table 1. CCS syntax

guage defined here. We refer the interested reader to [19] for operational semantics of CCS.

CCS provides a small set of operators that can be used to create systems with subsystem descriptions. Basic elements of CCS processes, as in most process calculi, are *actions*. Intuitively, actions represent atomic computational steps, that can be internal or external. All the former are identified by the silent action `tau`, while the latter are input/output operations on *channels* (i.e. communication ports), and represent potential interactions with the external environment. Another ingredient of CCS are the *constants*, that permit assigning symbolic names to CCS terms, and may appear in the terms themselves to specify recursive behaviours. In TAPAs, channels and constants names are sequences of alphanumeric characters (including the symbol _), where the former starts with a lowercase letter, and the latter with an uppercase letter. In the syntax definitions, we will use c to range over channels, K to range over constants, and C to denote sets of channels. Let c be a channel, CCS actions, ranged over by α, are defined by the following grammar:

$$\alpha \ ::= \ ?c \quad | \quad !c \quad | \quad \texttt{tau}$$

The meaning of actions is as follows: $?c$ is an input action on channel c, $!c$ is an output action on channel c, and `tau` represents internal computational actions.

The syntax of standard CCS terms is given by the grammar in the left part of Table 1, and it is used to define the CCS syntax adopted by the tool, given in the right part of the same table. Of these syntaxes, we only need to explain the term `let` D `in` S. It permits explicitly defining the CCS term declarations D called (directly or indirectly) by the process system S. The latter can be obtained by composing constants with static operators (i.e. parallel composition, relabelling and restriction, so called because, differently from choice and prefizing, they survive action transitions).

2.2 Graphical specification of terms

TAPAs allows user to specify processes not only as CCS terms but also by means of a graphical representation; in this section, we introduce the used graphical formalism.

TAPAs editor provides two separate windows (which contain different work tools), where users can draw separately processes and process systems (see Figure 4). Generally, the graphical representation of processes is independent from a specific process algebra, except for the actions notation. A TAPAs process is the representation of a CCS term declaration of the form $K = T$ (see the right part of Table 1); in the graphical notation the term T is represented as a graph, while the process name is written on the initial state of the graph (see Figure 5). The `nil` operator, i.e. the CCS empty process, is represented as a red circle with a black 'X' (see Figure 5) and denotes a *deadlock state*.

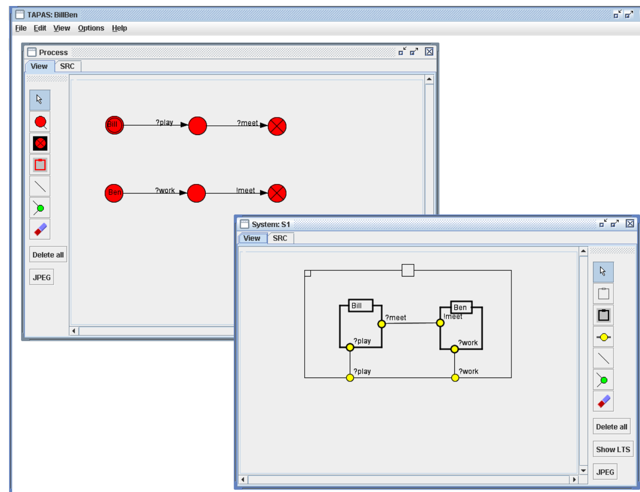

Fig. 4. TAPAs editor

The *action prefix* operator is rendered as a labelled edge which links two states; the edge is labelled with the corresponding action. For example, the process $P = ?a.\mathtt{nil};$ is represented as shown in Figure 5.

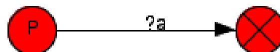

Fig. 5. Graphical representation of process $P = ?a.\mathtt{nil};$

Process names can be represented in two different ways, depending on the kind of the invoked term. If it is a LTS term, the name is drawn as a labelled state; if the term is a process system, then the corresponding graphical object is a red rectangle with inside the name associated to the invoked term. For example, the graphical representation of the process $P = !a.Q$ depends on the type of Q: if Q is a LTS term then we have the graphical representation shown on the left side of Figure 6, otherwise we have that on the right side.

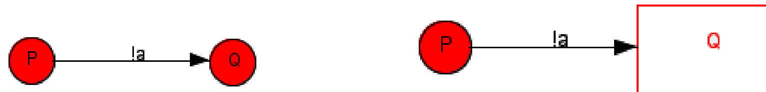

Fig. 6. Graphical representation of constant calls

The *sum* (or choice) operator is represented with a set of edges outgoing from a single state. For instance, the graphical representation of the process

$$P = !a.\mathtt{nil} + !b.\mathtt{nil} + !c.P;$$

is shown in Figure 7.

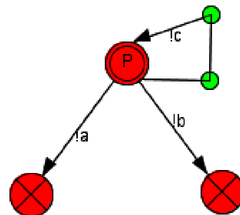

Fig. 7. Graphical representation of the choice operator

Let us consider how *process systems* are graphically represented. The basic idea is to represent process systems (and the processes that occur inside them) as boxes, where for each action there is, in the border of the corresponding box, a port (represented as a yellow circle) labelled with the action name. We can have two different kind of boxes:

1. *Process box:* it permits including a process into a process system, and it cannot contain another boxes into itself. It is drawn as a rectangle with a thick black border. When a user creates a process box, TAPAs automatically sets the process name as

the label of the box, and inserts the available ports on the box border. These ports correspond to the actions that the process can perform. For example, the box for the process $P = !a.?b.\mathtt{nil} + !c.?b.\mathtt{nil}$; is shown in the left side of Figure 8.

2. *System box:* it permits representing *parallel composition* of processes or process systems, because it can contain other boxes. In particular, it is possible to nest system boxes, in order to define complex system specifications. A system box is drawn as a rectangle with a thin gray border, and a user can define its label (see the box in the right side of Figure 8). It can also be used to represent *relabelling* and *restriction* operators by having actions on the border or inside the box.

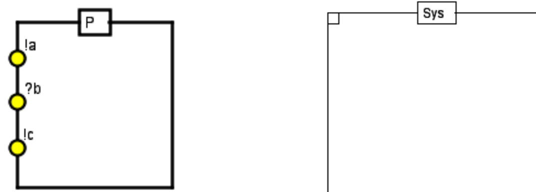

Fig. 8. Graphical representation of a process box and a system box

Before going into details of process systems, we need to introduce some auxiliary definitions:

– if a box F contains a box C, then we say that F is the *father* of C;
– two (or more) boxes are *brothers* if either they have the same father or all of them do not have a father.

In TAPAs, two (or more) processes/process systems are composed in parallel if the corresponding boxes are brothers. For example, to compose in parallel two process $P1$ and $P2$ we have to put them in a system box, as shown in Figure 9.

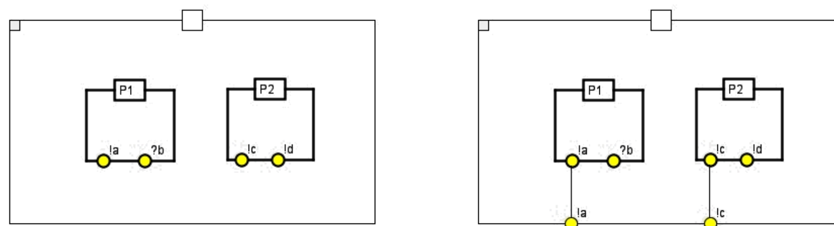

Fig. 9. Graphical representations of parallel composition of processes

When in a box a new port p is created (automatically or by a user), TAPAs looks for, among the box's brothers, ports that can synchronize with p, i.e. non-restricted ports labelled with the complementary action of that of p. For each port that can synchronize with p, TAPAs creates automatically a *synchronization link* between the ports. Notably, users cannot directly delete or create synchronization links. He/She needs to renaming/restriction in order to enable/disable process synchonization. For example, Figure 10 shows the graphical representation of the following system:

$$
\begin{array}{l}
\texttt{let} \\
\;\; P1 \;= !a.\texttt{nil}; \\
\;\; P2 \;= ?a.\texttt{nil}; \\
\texttt{in}\,(P1 \mid P2)\backslash\{a\};
\end{array}
$$

A port in a box is visible only from its brothers; in others words, putting a box B into another box produces the effect of restricting all the actions that B can perform. If a user wishes to make visible a restricted port p, he has to create a *relabelling link* from p to a port with the same label of the father box. For example, in the system shown on the left side of Figure 9 all the actions/ports of processes $P1$ and $P2$ are restricted. Instead, in the system shown on the right side of Figure 9, the actions $!a$ and $!c$ are visible, while $?b$ and $!d$ are restricted.

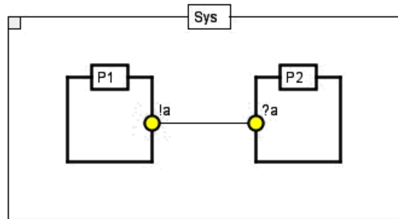

Fig. 10. A system with a synchronization on channel a

All in all, differently from other similar tools, that are tied to a single process algebra, the TAPAs graphical formalism is suitable for representing terms of different process algebras.

3 Verification of process properties

The data structures generated by the runtime environment, i.e. the labelled transition systems, can be used by other TAPAs components to analyze the corresponding concurrent systems.

3.1 Equivalence checking and graph minimization

The *Equivalences checker*, given two LTSs, permits verifying different kind of equivalences between them. It is worth noting that if other process algebras (e.g. value-passing

CCS, CSP,...) will be added to TAPAs, their integrations with the equivalence checker will be quite seamless.

At this moment, TAPAs considers two kind of equivalences:

1. bisimulations (strong, weak and branching) [19, 24];
2. decorated trace equivalences [14, 9].

For the last category, TAPAs provides the following equivalences:

– strong and weak trace;
– (weak) completed trace;
– divergence sensitive weak (completed) trace;
– must;
– testing;

Decorated trace equivalences have been implemented by combining a set of flags, which enable or disable some properties (see Figure 11). Flags and their meanings are as

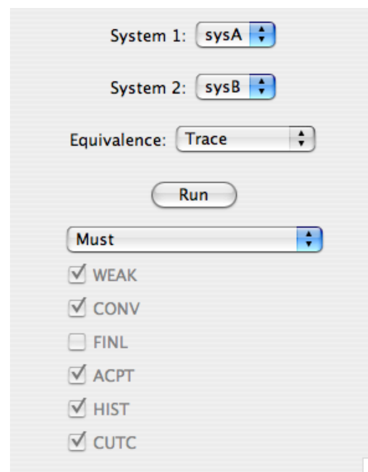

Fig. 11. Equivalence checker panel

follows:

– WEAK: flag for weak equivalences;
– CONV: flag for convergence sensitive trace equivalences;
– FINL: flag for equivalences sensitive to final states;
– ACPT: flag for equivalences sensitive to acceptance sets;
– HIST: flag for equivalences that consider histories;
– CUTC: flag for equivalences that ignore behaviour after divergent nodes.

For example, the weak trace is obtained by enabling only the WEAK flag, the completed trace by enabling the FINL flag, and the weak completed trace by enabling the WEAK and the FINL flags.

Finally, TAPAs minimizer allows users to minimize LTSs with large number of states, preserving strong, weak or branching bisimulation. This feature is obtained by using the equivalence checker algorithms.

3.2 Model checking

TAPAs can be used to analyze concurrent systems also by verifying satisfaction of properties, expressed as logical formulae.

For efficiency issues, the *model checker* takes in as an input only μ-calculus formulae [16]. However, TAPAs can accepts also other logics that permit specifying properties in more friendly ways. The formulae of these logics are translated in μ-calculus formulae and the verifications are performed by using the algorithm for the model checking of μ-calculus.

At the moment, besides the μ-calculus, TAPAs accepts only one other logic: ACTL (Action Computation Tree Logic [10]), an action based version of the branching time logic CTL [8], that is more expressive than Hennessy-Milner Logic and can naturally describe safety and liveness properties. To consider other logics it is however only needed to define a tranlation function from its formulae to μ-calculus formulae.

4 The study of a mutual exclusion algorithm

In this section we present how TAPAs can be used to deal with a full, althought simple, example. We shall consider one of the solution proposed to tackle the so called *mutual exclusion problem*. Mutual exclusion algorithms are used in concurrent programming to avoid that pieces of code, called *critical sections*, simultaneously access a common resource, such as a shared variable. We consider the Peterson's algorithm, that allows two processes to share a single-use resource without conflict. The two processes, $P1$ and $P2$, are defined by the following symmetrical pieces of pseudocode:

$P1$

```
while true do {
  <noncritical section>
  B1 = true;
  K = 2;
  while (B2 and K==2) do skip;
  <critical section>
  B1 = false;
}
```

$P2$

```
while true do {
  <noncritical section>
  B2 = true;
  K = 1;
  while (B1 and K==1) do skip;
  <critical section>
  B2 = false;
}
```

The two processes communicate by means of three shared variables, $B1$, $B2$ and K. The first two are boolean variables and hold true when the corresponding process wants

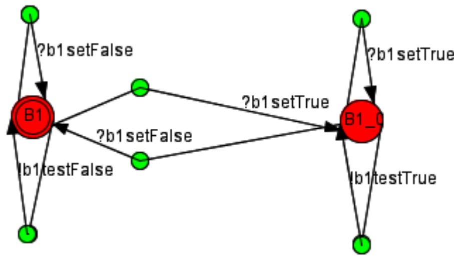

Fig. 12. Process $B1$

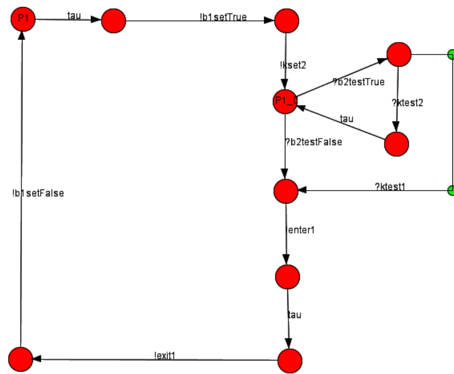

Fig. 13. Process $P1$

to enter the critical section. The last variable holds the identifier of the process (i.e. 1 or 2) whose turn it is. The algorithm guarantees the mutual exclusion property: $P1$ and $P2$ can never be in the critical section at the same time.

The three variables can be easily modelled in TAPAs as two-states processes, where each state represents a value that the variable can assume, as shown in Figure 12 for variable $B1$ (variables $B2$ and K can be rendered in a similar way).

Similarly, also processes $P1$ and $P2$ can be modelled as TAPAs processes. Since the processes are symmetric, Figure 13 shows only one of them (i.e. $P1$).

Finally, the whole process system, depicted in Figure 14, is created by putting the above five processes in parallel and by restricting the synchronization ports, and has the following textual representation:

$$\begin{aligned} Sys \;=\; & (B1 \mid B2 \mid K \mid P1 \mid P2) \setminus \{ ktest1, kset2, ktest2, b1setFalse, b1testFalse, kset1, \\ & b2testFalse, b2testTrue, b1testTrue, b1setTrue, b2setTrue, b2setFalse \}; \end{aligned}$$

The process system can interact with the external environment only by means of channel $enter1$, $enter2$, $exit1$ and $exit2$, which indicate entering and exiting of the two processes from the critical sections.

Using TAPAs we can verify some properties of the system. By means of the equivalence checker, a user can test the equivalences between the system's implementation

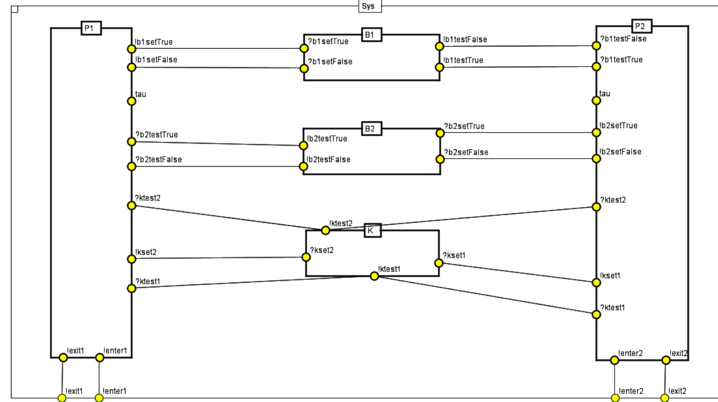

Fig. 14. The process system Sys

and the mutual exclusion specification shown in Figure 15. The latter process, named *Spec*, models the cyclical behaviour of entering and exiting of $P1$ and $P2$ (without distinction between them) from their critical sections, so that they can never be in the critical sections at the same time (indeed, the process cannot perform two consecutive actions *enter*).

Notably, at this level of abstraction it is not necessary to specify the identifier of the process that enters the critical section. Thus, before executing the test, the process system *Sys* must be slightly modified as follows:

$$Sys\ [enter/enter1, enter/enter2, exit/exit1, exit/exit2]$$

Of course, the tailored process *Sys* and process *Spec* are weak bisimilar. However, they are not testing equivalent, because, due the busy-waiting, the system implementation can diverge.

Another way to specify and to prove properties of the system *Sys* is by using the TAPAs model checker. For instance, by using the tool a user can verify that the Peterson's algorithm enjoys the following relevant properties specified in μ-calculus:

- deadlock-freedom: in each state, the system can perform at least an action

$$\nu X.\langle - \rangle \mathtt{true} \wedge [-]X$$

- livelock-freedom: the system cannot reach a state where it can perform only infinite sequences of internal actions;

$$\neg \mu X.\langle - \rangle X \vee \nu Y.[-\tau]\mathtt{false} \wedge \langle \tau \rangle \mathtt{true} \wedge [\tau]Y$$

- starvation-freedom: if a process wants to enter its critical section, eventually it succeeds.

$$\mu X.[-]X \vee \langle \mathtt{!enter\ i} \rangle \mathtt{true}$$

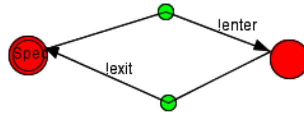

Fig. 15. Mutual exclusion specification

5 Concluding remarks

We have introduced TAPAs, a tool for the specification and the analysis of concurrent systems. TAPAs does not aim to compete with other tools, such as CWB [20], CWB-NC [2], UPPALL [4], CADP [12], EST [17], etc., in term of performance. TAPAs is thought of as a teaching support for Theory of Concurrency lectures. Indeed, the major feature that distinguish TAPAs from the other tools is its independence from a specific process algebra (and logic), that is also witnessed by its generic graphical formalism.

By comparing the lectures where TAPAs was exploited as teaching support with the 'classical' ones, we have noticed that the students got interested more in the subject. The students that have developed simple (but realist) case studies using TAPAs, have shown a deeper understanding of process algebras, behavioural equivalences and model checking.

As a future work, we plan to continue our development of the tool, by adding modules to deal with other process algebras, such as value-passing CCS [19], Lotos [23], CSP [7], etc., and by allowing users to specify properties of systems by other logics. Moreover, we will add other analysis tools, such as simulator that allows "animating" (concurrent components of) the system to look for deadlocks or livelocks. Finally, we plan to improve the usability of our equivalence checker, in order to return, in case of an unsuccessful verification, a counterexample, that is a property satisfied by only one of the two analyzed processes, and is expressed as a logical formula, e.g. a Hennessy-Milner Logic formula [13]. Thanks to TAPAs modularity, the different components can be developed independently.

Acknowledgements. We would like to thank Fabio Collini, Massimiliano Gori, Stefano Guerrini and Guzman Tierno for having contributed with their master theses to the development of key parts of the software at the basis of TAPAs.

References

1. TAPAs: a Tool for the Analysis of Process Algebras.
   Web site: `http://rap.dsi.unifi.it/tapas`.
2. R. Alur and T. Henzinger. The ncsu concurrency workbench. In *Proceedings of Computer-Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397. Springer–Verlag, 1996.
3. J.C.M. Baeten and W.P. Weijland. *Process algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

4. G. Behrmann, A. David, and K.G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, 2004.
5. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
6. H. Bowman and R. Gomez. *Concurrency Theory: Calculi an Automata for Modelling Untimed and Timed Concurrent Systems.* Springer, 2006.
7. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
8. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.
9. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.
10. R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In *Proc. Ecole de Printemps on Semantics of Concurrency, LNCS 469*, pages 407–419, 1990.
11. W. Fokkink. *Introduction to Process Algebra.* Springer, 2000.
12. H. Garavel, F. Lang, and R. Mateescu. An overview of cadp 2001. In *European Association for Software Science and Technology (EASST)*, volume 4 of *Newsletter*, pages 13–24, 2002.
13. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.
14. C.A.R. Hoare. A Model for Communicating Sequential Processes. In *On the Construction of Programs*, pages 229–254. Cambridge University Press, 1980.
15. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
16. D. Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
17. R. Meolic, T. Kapus, and Z.Brezocnik. The efficient symbolic tools package. In *Proceedings of the 8th International Conference Software, Telecommunications and Computer Networks (SoftCOM 2000)*, 2000.
18. R. Milner. *A Calculus of Communicating Systems.*, volume 92 of *Lecture Notes in Computer Science*. Springer–Verlag, 1980.
19. R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
20. F. Moller and P. Stevens. Edinburgh Concurrency Workbench user manual. Available from http://homepages.inf.ed.ac.uk/perdita/cwb/.
21. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
22. S.A. Schneider. *Concurrent and Real-Time Systems: The CSP Approach.* Wiley & Sons, 1999.
23. P.H.J. van Eijk, C.A. Vissers, and M. Diaz. *The formal description technique LOTOS.* Elsevier Science Publishers B.V., 1989.
24. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.