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
Viale Morgagni 65, 50134 Firenze, Italia
{calzolai,denicola,loreti,tiezzi}@dsi.unifi.it

Abstract. Process Algebras are formalisms for modelling concurrent 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. It does not aim at guaranteeing high performances, but has been developed as a support to teaching. 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, that makes it particularly suitable for teaching, is that it maintains a consistent double representation of each system both as a term and as a graph. Another useful didactical feature is the exhibition of counterexamples in case equivalences are not verified or the proposed formulae are not satisfied.

Key words: Concurrency, property verification, process algebras, bisimulation, behavioural equivalences, modal logics.

1 Introduction

Process algebras are a set of mathematically rigorous languages with well defined semantics that permit describing and verifying 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 constructors for system descriptions and are equipped with an operational semantics that describes systems evolution. Moreover, they often come equipped with observational mechanisms that permit identifying (through behavioural equivalences) those systems that cannot be taken apart by external observations. In some cases, process algebras have also complete axiomatizations, that capture the relevant identifications.

There has been a huge amount of research work on process algebras carried out during the last 25 years that started with the introduction of CCS [18, 19], CSP [6] and ACP [4]. In spite of the many conceptual similarities, these process algebras 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 the theoretical version of a practical language for concurrency and is still based on an operational intuition which, however, is interpreted w.r.t. a more abstract theory of decorated traces. ACP started from a completely different viewpoint and provided a purely mathematical algebraic view of concurrent systems: ACP processes are the solutions of systems of equations (axioms) over the signature of the considered algebra; operational semantics and bisimulation (in this case a different notion of bisimulation - 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, their 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, 22, 23], ACP [3, 11], Lotos [5] 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 tool we describe in this paper aims at supporting such courses.

The main ingredients of a specific process algebra are:

1. A minimal set of well thought operators capturing the relevant aspect of systems behavior and the way systems are composed.
2. A transition system associated with 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 with 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 some temporal logic's formulae [16, 7].

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 tree of relevant actions the system 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 with terms via a set of structural operational semantics rules and model checking is used to determine whether the transition systems associated with those terms enjoy the property specified by the given formulae.

In both approaches Labelled Transition Systems (LTSs) 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 relation describes systems evolution 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 LTSs.

LTSs are also the central ingredient of TAPAs, the software tool that we have implemented to support teaching of process algebras. Indeed, the main components of TAPAs are those permitting to minimize LTSs, to test their equivalence and to model check their satisfaction of temporal formulae. By relying on a sophisticated graphical user interface TAPAs permits:

- Understanding the meaning of the different process algebras operators by showing how these operators can be used to compose terms and the changes they induce on the composed transition systems.
- 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.
- Evaluating the different behavioural equivalences by having them on a single platform and checking the different equivalences by simply pushing different buttons.
- 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 front-end of TAPAs and show how it can be used for specifying behaviours of concurrent systems. In Section 3, we describe the components that can be used for verifying systems behaviours. In Section 4, we consider a more elaborate case study dealing with mutual exclusion algorithm. The final section contain a few concluding remarks and gives a brief account of related tools.

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 teaching Theory of Concurrency in a course of the Computer Science curriculum at 'Università degli Studi di Firenze'. TAPAs architecture is outlined in Figure 1. It consists of five components: an editor, a runtime environment, a model checker, an equivalence checker and a minimizer. TAPAs editor permits specifying concurrent systems as terms of a process algebra: terms can be inserted into the system by using either a textual representation or a graphical notation. The runtime-environment permits generating the

[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.

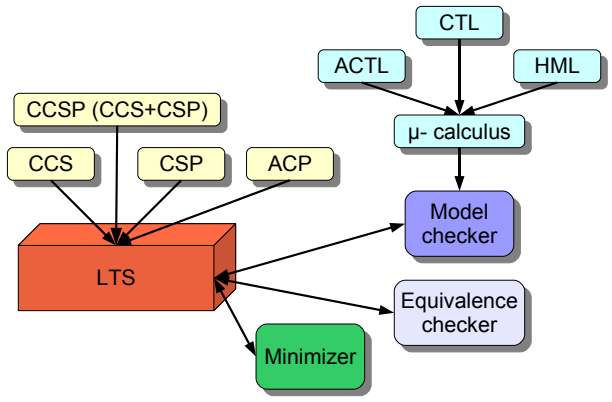

Fig. 1. TAPAs Architecture.

Labelled Transition System corresponding to a given specification. 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 whether two implementations of the same system are equivalent or not. Finally, the minimizer permits reducing the size of Labelled Transition Systems with large number of states while preserving the intended behaviour.

In TAPAs concurrent systems are described by means of *processes*, which are non-deterministic descriptions of system behaviours, and *process systems*, which are obtained by process compositions. Notably, processes can be defined in terms of other processes or other process systems. Processes and process systems are composed by means of the operators of a given process algebra. For instance, in the case of CCS, a process system can be obtained by parallel composition with binary synchronization, relabelling and restriction of processes, while in case of CSP a process system can be also obtained by using parallel composition with multi-party synchronization, internal and external choice operators and hiding.

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 guarantees the synchronization between the graph and the corresponding term. TAPAs does not rely on specific process algebra to be used for the systems specification. Currently, we are using CCSP[2] a process algebra obtained from CCS by incorporating some operators of CSP. However, thanks to the modular implementation of TAPAs, other process algebras can be easily added. Specifically, adding a new process algebra to TAPAs requires developing two JAVA packages: one for mod-

[2] Although the name is borrowed from [21] our variant is slightly different from the one considered by Olderog (op. cit.) and the one proposed by van Glabbeek and Vaandrager [24] due to the different mix of operators.

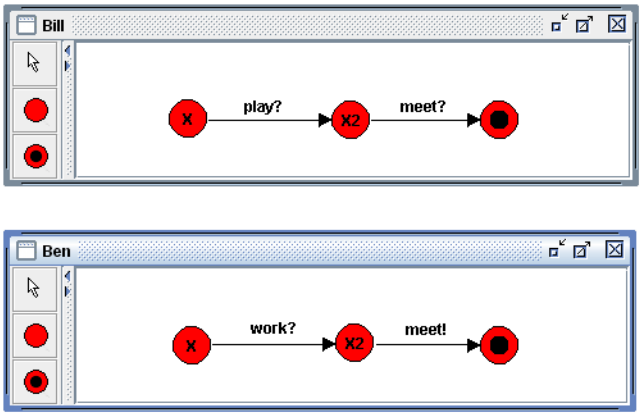

Fig. 2. Processes `Bill` and `Ben`.

elling the operational semantics of the operators and the other for defining the graphical representation of the operators.

Figure 2 shows two TAPAs processes that are the graphical representations corresponding to the following CCSP processes:

```
process Bill:
    X1 = play?.Bill[X2]
    X2 = meet?.nil
end
```

```
process Ben:
    X1 = work?.Ben[X2]
    X2 = meet!.nil
end
```

Process `Bill` can perform an input on channel `play` and continue with an input on `meet`, while process `Ben` can perform first an input on `work` and then an output on `meet`.

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. The process system corresponding to the parallel composition of the processes `Bill` and `Ben` is shown on the left hand side of Figure 3. To guarantee synchronization between `Bill` and `Ben`, channel `meet` is restricted. This is represented graphically by a black barrier around parallel composition. The textual representation of `BillBen` process system is the following:

```
system BillBen:
    restrict {meet} in
        Bill[X1] | Ben[X1]
    end
end
```

The LTS generated by the runtime component, corresponding to the above process system is reported on the right hand side of Figure 3. To help the user to analyze the generated graph, TAPAs provides visualization algorithms for drawing LTSs; new algorithms for drawing graphs can be easily plugged into TAPAs. When a LTS is generated starting from a system process, TAPAs will check satisfaction of the syntactic condi-

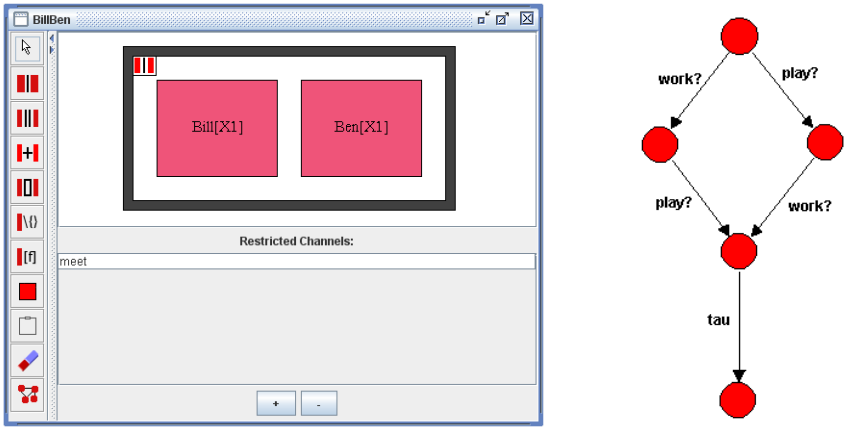

Fig. 3. Process system `BillBen` and the corresponding LTS.

tions that guarantee finiteness of the generated graph. When finiteness is not guaranteed, a warning message is displayed.

2.1 Textual specification of terms

The TAPAs runtime environment takes as input a textual specification, written in some process algebra, and generates the corresponding LTS, which can be used by the other components for model and equivalence checking. Currently, the only process algebra that can be used to specify concurrent systems with TAPAs is CCSP. Its set of operators is not intended to be minimal. Redundancy is tolerated for making it easier to specify systems specifications while keeping them understandable. In this section, we present the syntax and the operational semantics of the CCSP terms accepted by the runtime environment.

Basic elements of CCSP processes, as in most process calculi, are *actions*. Intuitively, actions represent atomic computational steps, that can be internal or external. All internal actions are rendered as the silent action `tau`, while external actions are input/output operations on *channels* (i.e. communication ports), and represent potential interactions with the external environment.

CCSP syntax. The syntax of a CCSP module is given in Table 1; there we have used $\sum_{i \in I} \mathsf{ACT}^i_j \,.\, \mathsf{PROC}^i_j$ for $\mathsf{ACT}^1_j \,.\, \mathsf{PROC}^1_j + \cdots + \mathsf{ACT}^n_j \,.\, \mathsf{PROC}^n_j$ if $I = \{1, \ldots, n\}$. The set of names $\mathcal{N} = \mathcal{PN} \cup \mathcal{XN} \cup \mathcal{CN} \cup \mathcal{SN}$ contains (non-empty finite) sequences of alphanumeric characters (including the symbol _) where:

- `P` ranges over the set of *process names* $\mathcal{PN}$,
- `X` ranges over the set of *state names* $\mathcal{XN}$,
- `c` ranges over the set of *channel names* $\mathcal{CN}$,
- `S` range over the set of *system names* $\mathcal{SN}$.

Table 1. CCSP syntax.

M	::=	PROC_DEC | SYS_DEC | M M	*(Module)*

PROC_DEC	::=	`process P :`
		$\mathtt{X}_1 = \sum_{i \in I_1} \mathsf{ACT}_1^i \,.\, \mathsf{PROC}_1^i$
		$\cdots$
		$\mathtt{X}_n = \sum_{j \in I_n} \mathsf{ACT}_n^j \,.\, \mathsf{PROC}_n^j$
		`end`
			(Process dec.)

ACT	::=	`tau` | `c!` | `c?`	*(Action)*
PROC	::=	`nil` | `P[X]` | S	*(Process)*
SYS_DEC	::=	`system S :` COMP `end`	*(System dec.)*
COMP	::=	C | C_1 `(+)` C_2 | C_1 `[]` C_2 | C_1 | C_2	*(Components)*
C	::=	PROC	*(Component)*
	|	`sync on` CS `in` C_1 | C_2 `end`	
	|	`rename [` F `] in` COMP `end`	
	|	`restrict` CS `in` COMP `end`	
CS	::=	* | $\{\mathtt{c}_1, \ldots, \mathtt{c}_n\}$	*(Channel set)*
F	::=	c/c' | F, F	*(Renaming fun.)*

A CCSP module is a sequence of process declaration and system declarations. Processes are defined by "$state\_name = \sum_{i \in I} action.process$", where an action can be the silent action `tau` (where `tau` $\notin \mathcal{CN}$), an output `c!` or an input `c?` on a channel `c`, while a process can be the empty process `nil` (which cannot perform any actions), a reference `P[X]` to the state `X` of the process `P` or a reference to a system `S`. Systems are defined as the composition via parallel operator (i.e. `|`), external and internal choice operators (i.e. `[]` and `(+)` respectively) of elements called *components* . These can be processes or the result of applying an operation (multi-synchronization operation `sync`, renaming operation `rename`, restriction operation `restrict`) processes. The multi-synchronization construct is inspired by the parallel operator of CSP and allows parallel components to synchronize on any channel of the specified set when all of them can perform the same action. Renaming and restriction are the standard CCS operators; the former permits changing channel names, while the latter is used for delimiting their scope. For multi-synchronization and restriction operations, we use the wildcard symbol * to indicate $\mathcal{CN}$, i.e. the set of all channel names.

CCSP operational semantics. CCSP operational semantics is defined only for *well-formed* modules, i.e. modules where all used states, processes and systems have corresponding declarations. Moreover, it is assumed that states and systems names are distinct, well-formedness check can be statically performed. CCSP semantics is provided relatively to a module M that contains the necessary definitions. It is described

Table 2. CCSP operational semantics w.r.t. module M.

$$(P_{ref})\ \frac{(\,\texttt{process P : ... X}_i = \sum_{j\in I} \mathsf{ACT}_i^j \,.\, \mathsf{PROC}_i^j \ldots \texttt{ end}\,) \in \mathsf{M}}{\mathsf{P}[\mathtt{X}_i] \xrightarrow{\mathsf{ACT}_i^k} \mathsf{PROC}_i^k}\ (k \in I)$$

$$(S_{ref})\ \frac{(\,\texttt{system S : COMP end}\,) \in \mathsf{M} \qquad \mathsf{COMP} \xrightarrow{\mu} \mathsf{COMP}'}{\mathsf{S} \xrightarrow{\mu} \mathsf{COMP}'}$$

$$(Broad_1)\ \frac{\mathsf{C}_1 \xrightarrow{\mu} \mathsf{C}' \qquad \mu \notin act(\mathsf{CS})}{\texttt{sync on CS in } \mathsf{C}_1 \texttt{ | } \mathsf{C}_2 \texttt{ end} \xrightarrow{\mu} \texttt{sync on CS in } \mathsf{C}' \texttt{ | } \mathsf{C}_2 \texttt{ end}}$$

$$(Broad_2)\ \frac{\mathsf{C}_2 \xrightarrow{\mu} \mathsf{C}' \qquad \mu \notin act(\mathsf{CS})}{\texttt{sync on CS in } \mathsf{C}_1 \texttt{ | } \mathsf{C}_2 \texttt{ end} \xrightarrow{\mu} \texttt{sync on CS in } \mathsf{C}_1 \texttt{ | } \mathsf{C}' \texttt{ end}}$$

$$(Broad_3)\ \frac{\mathsf{C}_1 \xrightarrow{\alpha} \mathsf{C}_1' \qquad \mathsf{C}_2 \xrightarrow{\alpha} \mathsf{C}_2' \qquad \alpha \in act(\mathsf{CS})}{\texttt{sync on CS in } \mathsf{C}_1 \texttt{ | } \mathsf{C}_2 \texttt{ end} \xrightarrow{\alpha} \texttt{sync on CS in } \mathsf{C}_1' \texttt{ | } \mathsf{C}_2' \texttt{ end}}$$

$$(Ren)\ \frac{\mathsf{COMP} \xrightarrow{\mu} \mathsf{COMP}'}{\texttt{rename [F] in COMP end} \xrightarrow{\mathsf{F}(\mu)} \texttt{rename [F] in COMP' end}}$$

$$(Res)\ \frac{\mathsf{COMP} \xrightarrow{\mu} \mathsf{COMP}' \qquad \mu \notin act(\mathsf{CS})}{\texttt{restrict CS in COMP end} \xrightarrow{\mu} \texttt{restrict CS in COMP' end}}$$

$$(Sync)\ \frac{\mathsf{C}_1 \xrightarrow{\alpha} \mathsf{C}_1' \qquad \mathsf{C}_2 \xrightarrow{\bar{\alpha}} \mathsf{C}_2'}{\mathsf{C}_1 \texttt{ | } \mathsf{C}_2 \xrightarrow{\texttt{tau}} \mathsf{C}_1' \texttt{ | } \mathsf{C}_2'}$$

$$(Inter_1)\ \frac{\mathsf{C}_1 \xrightarrow{\mu} \mathsf{C}_1'}{\mathsf{C}_1 \texttt{ | } \mathsf{C}_2 \xrightarrow{\mu} \mathsf{C}_1' \texttt{ | } \mathsf{C}_2} \qquad (Inter_2)\ \frac{\mathsf{C}_2 \xrightarrow{\mu} \mathsf{C}_2'}{\mathsf{C}_1 \texttt{ | } \mathsf{C}_2 \xrightarrow{\mu} \mathsf{C}_1 \texttt{ | } \mathsf{C}_2'}$$

$$(Int.\ choice_1)\ \mathsf{C}_1(+)\mathsf{C}_2 \xrightarrow{\texttt{tau}} \mathsf{C}_1 \qquad (Int.\ choice_2)\ \mathsf{C}_1(+)\mathsf{C}_2 \xrightarrow{\texttt{tau}} \mathsf{C}_2$$

$$(Ext.\ choice_1)\ \frac{\mathsf{C}_1 \xrightarrow{\alpha} \mathsf{C}'}{\mathsf{C}_1 [] \mathsf{C}_2 \xrightarrow{\alpha} \mathsf{C}'} \qquad (Ext.\ choice_2)\ \frac{\mathsf{C}_2 \xrightarrow{\alpha} \mathsf{C}'}{\mathsf{C}_1 [] \mathsf{C}_2 \xrightarrow{\alpha} \mathsf{C}'}$$

$$(Ext.\ choice_3)\ \frac{\mathsf{C}_1 \xrightarrow{\texttt{tau}} \mathsf{C}'}{\mathsf{C}_1 [] \mathsf{C}_2 \xrightarrow{\texttt{tau}} \mathsf{C}' [] \mathsf{C}_2} \qquad (Ext.\ choice_4)\ \frac{\mathsf{C}_2 \xrightarrow{\texttt{tau}} \mathsf{C}'}{\mathsf{C}_1 [] \mathsf{C}_2 \xrightarrow{\texttt{tau}} \mathsf{C}_1 [] \mathsf{C}'}$$

as a labelled transition relation $\xrightarrow{\mu}$ over components induced by the rules in Table 2, where μ is generated by the following grammar:

$$\mu \ ::= \ \texttt{tau} \ \mid \ \alpha \qquad\qquad \alpha \ ::= \ \texttt{c!} \ \mid \ \texttt{c?}$$

The meaning of labels is the following: `tau` represents internal computational steps, while `c!` and `c?` denote execution of output and input actions on channel `c`, respectively. An input and output on the same channel are called *complementary labels*. We will use $\bar{\alpha}$ to denote the complement of α (i.e. $\overline{\texttt{c!}} = \texttt{c?}$ and $\overline{\texttt{c?}} = \texttt{c!}$), and $act(\mathsf{CS})$ to denote the set of actions corresponding to the channels of CS (i.e. $\texttt{tau} \notin act(\mathsf{CS})$, while $\texttt{c!}, \texttt{c?} \in act(\mathsf{CS})$ if either $\texttt{c} \in \mathsf{CS}$ or $\mathsf{CS} = *$).

Rule (P_{ref}) states that process `P[X]` evolves by performing one of the actions that `P` can execute from state `X`; the actual choice is nondeterministic. A system name evolves according to the actions of the body of the corresponding declaration, rule (S_{ref}). Rules for renaming, restriction, parallel composition, internal and external choice are standard (see [18] and [6]). Finally, rules $(Broad_1)$ and $(Broad_2)$ permit the interleaving of the actions of parallel components when actions outside the specified channel set are performed, rule $(Broad_3)$ allows multiple synchronization of processes on one of the synchronization channels.

2.2 Graphical specification of terms

In this section, we present the graphical formalism used for defining *processes* and *prosess systems*. TAPAs editor provides two separate kind of windows that can be used to draw processes and process systems (see Figure 4).

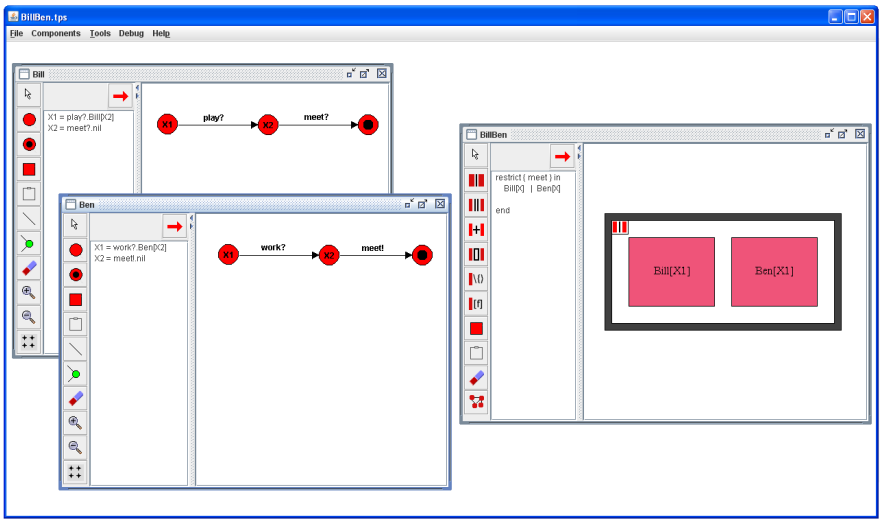

Fig. 4. A TAPAs screenshot.

Generally, the graphical representation of processes is independent from a specific process algebra, except for the labels corresponding to the actions of the algebras. A

process is rendered as a graph; its edges describe the performed actions and their effect, while its nodes represent systems configurations. We have four kinds of nodes:

1. **Terminal:** identify a terminal state (e.g. the empty process `nil`) and are represented as a red circle with a black dot.
2. **State Reference:** identify states defined within the considered process, and are represented as a red circle. Only this kind of nodes can have outgoing edges.
3. **Process Reference:** identify states defined in another processes, and are represented as a red box.
4. **System Reference:** identify systems, and are represented as a white box.

Figure 5 shows the graphical representation of the process below, if `P1` is a process and `Sys` is a system.

```
process P2:
    start = a?.nil + b?.P2[X1] + c?.P1[X] + d?.Sys
    X1 = e!.nil
end
```

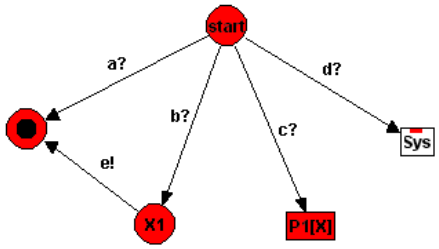

Fig. 5. Graphical representation of TAPAs nodes.

Process systems are graphically represented via nested boxes; each box represents either one of the system operators or a reference to a process or to a process system. For the sake of clarity, each system operator has a specific graphical box.

Figure 6 reports the graphical representation of the following process system:

```
restrict {a,b} in
  R[Z] [] P[X]
end
|
sync on {a,c} in
  Q[Y] | Sys
end
```

The outermost enclosing box represents the top level operator that, in this case, is parallel composition while its arguments are drawn as inner boxes. There are two inner components, one is a restriction the other is a multi-synchronization. Restriction is rendered

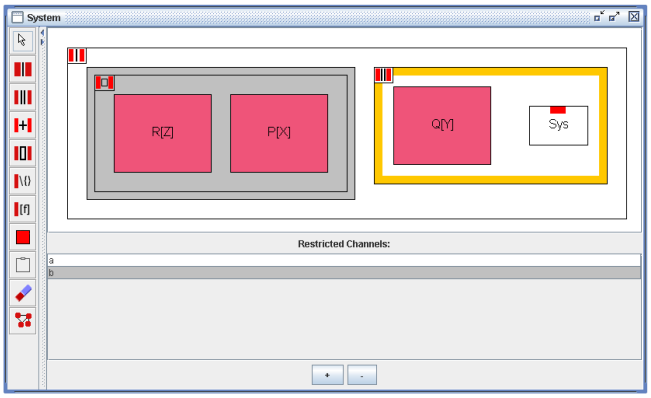

Fig. 6. Graphical representation of a process system.

as a box surrounded by a black barrier and contains an external choice between processes `R[Z]` and `P[X]`. Multi-synchronization is rendered as a box with a yellow frame that contains process `Q[Y]` and system `Sys`. When a box is selected, other parameters of the corresponding operator, such as restricted names, are shown in a separate table.

3 Verification of process properties

The LTS generated by the runtime environment can be used by the other TAPAs components to analyze the corresponding concurrent systems. The analysis can be performed either by an equivalence checker or by a model checker.

The TAPAs *Equivalences Checker* permits verifying different kind of equivalences between pairs of systems. It is worth noting that, if other process algebras (e.g. value-passing CCS, ACP, ...) were to be added to TAPAs, their integrations with the equivalence checker would be seamless.

Currently, TAPAs permits checking two kinds of equivalences:

1. Bisimulations based equivalences (strong, weak and branching) [19, 26];
2. Decorated trace equivalences (weak and strong variants of trace completed trace, divergence sensitive trace, must, testing) [14, 9].

Decorated trace equivalences have been implemented by combining a set of flags, which enable or disable checking specific properties (see Figure 7 left side). Flags, and their meanings, are the following:

- WEAK: weak equivalences;
- CONV: convergence sensitive equivalences;
- FINL: equivalences sensitive to final states;
- ACPT: equivalences sensitive to acceptance sets;
- HIST: equivalences that consider past divergences as catastrophic;
- CUTC: equivalences that ignore all behaviours after divergent nodes.

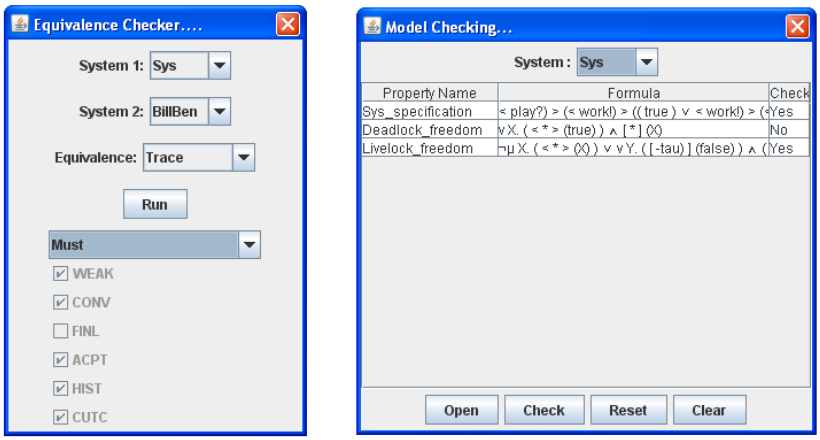

Fig. 7. Equivalence and Model Checker panels.

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

Whenever an equivalence check turns out to be unsuccessful, TAPAs provides counterexamples, i.e. evidences that the analyzed systems do behave differently. Hennessy-Milner Logic [13] formulae that capture a property satisfied only by one of the two in-equivalent processes are exhibited.

Equivalence checker algorithms are also used for implementing a LTSs *Minimizer*. This module allows users to minimize LTSs with large number of states while preserving strong, weak or branching bisimulation.

TAPAs can be used to analyze concurrent systems also by verifying satisfaction of properties, expressed as logical formulae. This task can be achieved by using the *Model Checker* that implements a Local Model Checking Algorithm [27], and permits verifying satisfaction of modal logic formulae by system processes (Figure 7 right side).

For efficiency reasons, the model checker takes as input only μ-calculus formulae [16]. However, TAPAs can be easily extended to accept also formulae from other logics like, for instance, ACTL (Action Computation Tree Logic [10]) that turns out to be more user friendly. Formulae of the new logics will have to be translated in equivalent μ-calculus ones and their verifications will be performed on the outcome of the translation.

4 The study of a mutual exclusion algorithm

In this section we present the *mutual exclusion problem*, one of the simpler examples that are used for supporting concurrency theory courses.

Mutual exclusion algorithms are used in concurrent programming to avoid that pieces of code, called *critical sections*, simultaneously access a common resource, such

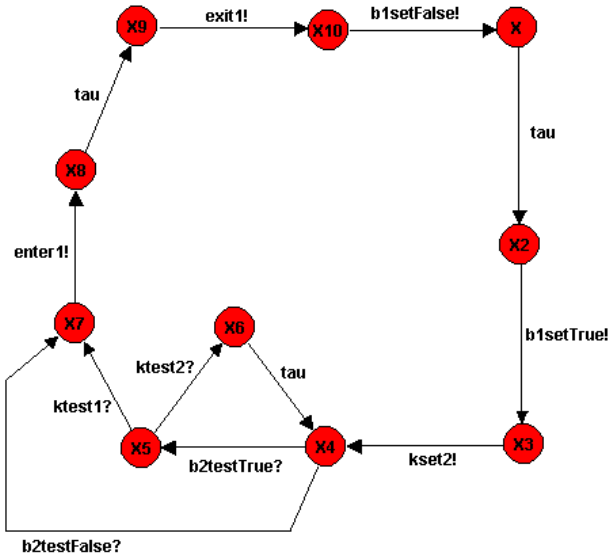

Fig. 8. Process P1.

as a shared variable. We consider Peterson's algorithm, that allows two processes to share a single-use resource without conflicts. 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 are true when the corresponding process wants to enter the critical section. The last variable contains the identifier of the process (i.e. 1 or 2) whose turn it is. The algorithm guarantees mutual exclusion: P1 and P2 can never be in their critical sections 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. Similarly, processes P1 and P2 can be modelled as TAPAs processes. Since the two processes are symmetric, Figure 8 shows only one of them (i.e. P1). The complete process system, reported in Figure 9, is obtained by putting the five processes in parallel and by restricting the syn-

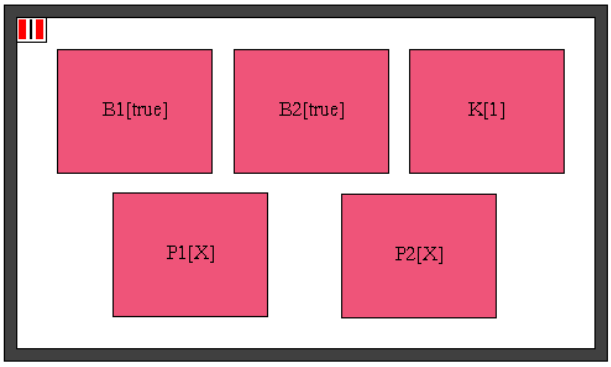

Fig. 9. The process system Sys.

chronization channels; it has the following textual representation:

```
system Sys:
    restrict { ktest1, kset2, ktest2, b1setFalse, b1testFalse,
               kset1, b2testFalse, b2testTrue, b1testTrue,
               b1setTrue, b2setTrue, b2setFalse } in
        B1[true] | B2[true] | K[1] | P1[X] | P2[X]
    end
end
```

Sys can interact with the external environment only by means of channels enter1, enter2, exit1 and exit2, that represent entering and exiting of the two processes from the critical sections.

Generally, after showing this example, we ask to students to try to find an alternative solution of the *mutual exclusion problem*. For instance we could ask to provide an alternative formalization that does not rely on shared variables. A possible solution is that based on the multi-synchronization operator. The algorithm that uses the multi-synchronization operator is reported in Figure 10.

There, enter1 and enter2 are synchronization channels; therefore the two processes P1 and P2 have to perform the same actions; if enter1! is the performed action, then P1 can enter its critical section and P2 must wait until P1 exits; if enter2! is the performed action, then P2 can enter its critical section and P1 must wait until P2 exits. This simple example is useful for showing that different primitives (multicast messages or singlecast messages) permit developing different solutions.

Using TAPAs students can verify properties of the systems. By means of the equivalence checker, equivalence of the system's implementation and the mutual exclusion specification reported in Figure 11 can be tested. Process Spec models the cyclical behaviour of entering and exiting of P1 and P2 (without distinction between them) from their critical sections. In this way it is specified that they can never be in the critical sections at the same time: two consecutive actions enter! cannot be performed. Notably,

```
process P1:
  X -> enter1!.P1[X2] + enter2!.P1[X3]
  X2 -> tau.P1[X4]
  X3 -> exit2!.P1[X]
  X4 -> exit1!.P1[X]
end
```

```
process P2:
  X -> enter2!.P2[X2]
       + enter1!.P2[X3]
  X2 -> tau.P2[X4]
  X3 -> exit1!.P2[X]
  X4 -> exit2!.P2[X]
end
```

```
system BroadSys:
    sync on {enter1, enter2, exit1, exit2} in
        P1[X] | P2[X]
    end
end
```

Fig. 10. The CCSP representation of the Mutual exclusion algorithm using broadcast.

at this level of abstraction it is not necessary to identify the actual process that is using its critical section. Thus, before executing the test, `Sys` and `BroadSys` must be slightly modified as follows:

```
rename [enter/enter1, enter/enter2, exit/exit1, exit/exit2] in
    Sys/BroadSys
end
```

The processes we have just modified and `Spec` are weakly bisimilar. However, due to busy-waiting, `Sys` is not testing equivalent to `Spec`, because `Sys` can diverge while `BroadSys` cannot.

The behaviour of the systems specified so far can also be verified through the TAPAs model checker. For instance, it can can be verified that the implementations of Peterson's algorithm and the one based on multi-synchronization enjoy the following relevant properties specified in μ-calculus [16]:

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

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

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

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

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

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

At the end of the academic course we assign to students a case study and they have to specify and verify it using TAPAs. We have noticed that, at first, students try to specify all the features of the system, even those redundant and not necessary. Often,

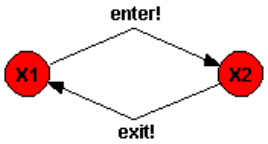

Fig. 11. Mutual exclusion specification.

after experiencing problems related to the state space explosion they understand the need of abstract description. Thus, they reduce the number of the states by simplifying the system omitting the unnecessary aspects, to capture only the interesting behaviour of the analized system. In some case they also use the minimization facility to reduce the size of the components before actually composing them to obtain systems..

5 Conclusions and Related Work

We have introduced TAPAs, a tool for the specification and the analysis of concurrent systems. TAPAs has been designed to support teaching concurrency and one of its distinguishing feature is the independence from specific process algebras and logics, that is guaranteed by its generic graphical formalism. TAPAs assigns a central role to Labelled Transition Systems. By considering the LTS associated to the different terms students can appreciate similarities and differences between the operators. Moreover, by studying the effect of some of the most important equivalences over LTS, students can appreciate their impact on specific calculi and gain insight into the nature of their nature.

By comparing the lectures where TAPAs was used as teaching support with the 'classical' ones, we have noticed that the students got significantly more interested in the subject. The students that have developed simple (but realistic) case studies using TAPAs, have shown a deeper understanding of process algebras, behavioural equivalences and model checking. In spite of its didactical nature, TAPAs has also been used to deal with more complex systems and we plan to use it to gently expose researchers from industry to the use of formal methods.

In the last years many other tools were developed, but, generally, they are not intended to support teaching: some tools have not a graphical user interface, others do not support the process algebras commonly used in the academic courses (CCS, CSP) and just few tools allow the graphical specification.

One of the most used tool for teaching concurrency, that follows a process algebraic approach, is LTSA [17]. It permits generating LTS starting from a term written in a simple process algebra (named FSP), but it does not allows a direct graphical specification of terms. LTSA allows the verification of systems properties by reachability analysis based on formulae of a Linear Time Temporal logic (named Fluent LTL), and it generates traces leading to failures whenever the specified property is not satisfied. Differently from TAPAs, LTSA does not provide an equivalence checker.

Another well-known tool for process algebras is CADP [12]: it offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking, but it does not allow the graphical specification and the systems descriptions have to be written in LOTOS [25] that is not a widely used process algebra.

CWB [20] and CWB-NC [2] are very efficient tools that permit specifying and verifying properties of concurrent systems. These tools support many process algebras and can be used for checking many behavioural equivalences. However, both CWB and CWB-NC do not provide a graphical interface that can be used for describing concurrent systems. Clearly, this can make difficult to use tools in academic course for introducing theory of concurrency.

As a future work, we plan also to continue the development by adding modules to deal with other process algebras, such as value-passing CCS [19] and LOTOS, and with other logics. Moreover, we will add other analysis tools, such as a simulator that allows "animating" the system showing the possible interactions between its components. We plan also to improve the TAPAs back-end in order to support systems with a larger state space. Moreover we plan to enrich TAPAs along the lines of PAC [8] to permit users to define their own operators and to generate the LTS associates to terms containing these new operators.

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. `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. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
5. H. Bowman and R. Gomez. *Concurrency Theory: Calculi an Automata for Modelling Untimed and Timed Concurrent Systems.* Springer–Verlag, 2006.
6. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
7. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of Logic of Programs*, pages 52–71. Springer–Verlag, 1982.
8. R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In *Proceedings of the Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019, pages 153–173. Springer–Verlag, 1995.
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 *Proceedings of the Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer–Verlag, 1990.

11. W. Fokkink. *Introduction to Process Algebra.* Springer–Verlag, 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. J. Magee and J. Kramer. *Concurrency: State Models and Java Programs.* John Wiley and Sons Inc, 2006.
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. Ernst-Rüdiger Olderog. Operational Petri net semantics for CCSP. In *Proceedings of European Workshop on Applications and Theory of Petri Nets*, volume 266, pages 196–223. Springer–Verlag, 1987.
22. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
23. S.A. Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. Wiley & Sons, 1999.
24. R.J. van Glabbeek and F.W. Vaandrager. Bundle event structures and CCSP. In *Proceedings of 14th International Conference on Concurrency Theory (CONCUR 2003)*, volume 2761 of *Lecture Notes in Computer Science*, pages 57–71. Springer–Verlag, 2003.
25. P.H.J. van Eijk, C.A. Vissers, and M. Diaz. *The formal description technique LOTOS.* Elsevier Science Publishers B.V., 1989.
26. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.
27. G. Winskel. Topics in concurrency. Lecture notes, University of Cambridge, 2008. Available from `http://www.cl.cam.ac.uk/ gw104/TIC08.ps`.