Proving *Testing Preorders* for Process Algebra Descriptions

Fulvio CORNO, Marco CUSINATO, Mario FERRERO, Paolo PRINETTO

Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy

Abstract*

Process Algebras are rapidly becoming a mathematical model used by verification engineers to extend the description capabilities of Finite State Machines towards higher abstraction levels. As long as design and verification methodologies at the system level are developed, the wide spectrum of equivalence relations that can be defined over processes receives an ever increasing importance. Testing Equivalences *and* Testing Preorders *are particularly suited for formalizing the relationships holding in top-down hierarchical methodologies. The main deterrent to the widespread use of Process Algebras seems to be the lack of efficient tools. Very efficient algorithmic techniques, based on the adoption of Binary Decision Diagrams, are now being used in different fields. This paper presents algorithms for the proof of testing preorders and equivalences that are, to the best of our knowledge, the first successful attempt to implement testing relations with BDDs. Experimental results show that the the implemented algorithms are able to deal with medium- and large-size systems.*

1. Introduction

Finite State Machines (FSMs) are one of the most popular formalisms for describing the behavior of systems. They proved an excellent formalism to describe and manipulate systems when synchronous descriptions are considered. Unfortunately, they lack versatility and expressive power to be used in higher-level descriptions, where more sophisticated models are needed. This lead to the development of several extensions of FSMs, in the directions of concurrence, non determinism, asynchronous evolution, hierarchy, communication, and data handling.

Process Algebras (PAs) [14] [12] [16] are one such extension, aiming at modeling concurrent communicating systems at a very high level of abstraction. They overcome most limits of FSMs, especially for control-dominated systems, and they profit by a sound mathematical framework. In formal verification, PAs support the powerful notion of *observational equivalences* and *preorders* [8], which allow the designer to select the resolution power he needs when comparing systems.

The main limit to the applicability of PAs is the lack of efficient tools. The algorithms presented in this paper have been implemented in SEVERO [4] [5], a BDD based tool for Process Algebra manipulation. Its efficiency is orders of magnitude superior to traditional approaches [6] [9] and comparable or better to other symbolic implementations [1]. The PA impemented in SEVERO is Circal [15] [17], whose capability of describing concurrent events and multi-point communications is invaluable in describing hardware systems.

The goal of this paper is twofold: from one hand, it presents *Testing Preorders* and *Equivalences*, showing how they can be used in system design and verification. From the other hand, it describes a set of symbolic algorithms to minimize processes according to Testing Equivalence and to check for testing relations.

The paper is structured as follows. Section 2 shortly establishes the theoretical framework of PAs and Testing Relations. Section 3 details the algorithms used in verification. Section 4 reports some experimental results proving the efficiency of the approach, while in section 5 some conclusions are drawn.

2. Theoretical Framework

2.1. Process Algebras

A *process* is a black box offering to the external environment a set of communication *ports*, through which *events* are exchanged. Following Circal's para-

* This work has been partially supported by the Esprit Project 6575 "ATSEC", Human Capital and Mobility EEC-HCM-0061 "EUROFORM: Formal Methods for Correct System Design" and by the Italian National Research Council "Progetto Speciale Specifica ad alto Livello e Verifica Formale di Sistemi Digitali". Contact address: Paolo Prinetto, Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino (Italy), e-mail Paolo.Prinetto@polito.it

digm [15], actions can occur simultaneously and can be shared among processes according to multi-point rendez-vous *channels*. When an external observer is not allowed to monitor this exchange of actions the processes are said to execute internal communications, called τ or *silent actions*.

The behavior of a process K is specified in terms of the events it is able to exchange and Labeled Transition Systems (LTSs) are used as a concrete representation. A LTS is a 4-tuple $K = (S, A, T, s_0)$, where S is a set of states, A is the set of *actions* that K is allowed to execute, where an action $a \in A$ is a set of events taken from the set L of allowed events: $A=2^L$, and s_0 is the initial state $s_0 \in S$. T is a relation $T \subseteq S \times A \times S$, called *transition relation*, that determines the sequential behavior of the process: it records the set of *transitions* $\langle ss, a, as \rangle$ that can be traversed by the system, where $ss, as \in S$ are the *start* and the *arrival state*, and $a \in A$ is an *action*.

Process Algebras provide operators for manipulating processes described by LTSs. Process Algebra operators are used to build composite systems starting from simpler ones [17]. The most important operators are:

- **Composition operators**: the concurrent evolution of two processes is defined by considering all the *interleavings* of their actions and the synchronizazions due to rendez-vous on *shared actions*, which model communications.
- **Choice operators**: the composite behavior is that of *one* of the components.
- **Abstraction operators** are defined to *hide* from an external observer the inner communications and to simplify processes according to the constraints imposed by their environment.

When Finite State Machines are considered, a simple notion of equivalence is used, namely *trace* (or *language*) equivalence. Unfortunately, trace equivalence is not resolutive enough to catch some important properties of concurrent and not deterministic processes, such as deadlock. Thus, more powerful notions of *observational equivalence* have to be defined.

A first approach to observational equivalence [14] defines two processes as *indistinguishable* if each one is able to *simulate* the behavior of the other one, i.e., if it is able to provide the same sets of events to the external world. Two such processes are said to be *bisimilar*. There are several possible definitions for bisimilarity, in terms of increasing resolution power, e.g., *weak*, *branching*, and *strong bisimilarity*. They differ mainly for the kind of internal actions each process is allowed to execute when trying to simulate the other.

An different approach to observational equivalence is that of *testing equivalence* or, in general, *testing relations* [8]. This approach is truly observational since two systems are said equivalent whenever they *satisfy* the same set of *observers*. Depending on the definition of the universe of observers and on the notion of satisfaction, different testing equivalences can be defined.

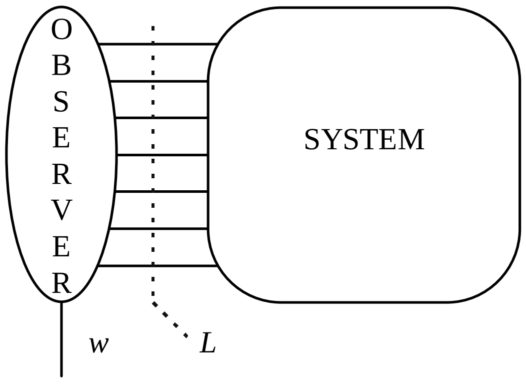

Figure 1: *Definition of an observer*

2.2. Testing Relations

Given a system K defined on a set of events L, an *observer* [7] O for K is a process defined over events $L \cup \{w\}$. w is an additional event ("victory") used by the observer to report to the external world the success of its observation. The observer and the system interact via their common events L (Fig. 1).

To define the satisfaction of an observer, one considers the process obtained by composing the system and the observer, i.e., the set of all possible common computation paths. Given an observer, a system *must satisfy* it if, for every path, the observer is satisfied, whereas a system *may satisfy* the observer if there exists at least one path in which the observer is satisfied. If none of these conditions holds, then the system *can not satisfy* the observer.

For example, in Fig. 2 a system K is defined by its LTS. By the above definitions, it turns out that:

- K must satisfy O_1 since in the process K^*O_1, obtained by parallel composition of the system and the observer, the event w is inevitable;
- K may satisfy O_2 since in K^*O_2 w is possible;
- K can not satisfy O_3 since K^*O_3 can't execute w.

In this framework, all the possible relations among processes are defined as relations between sets of observers. In particular:

- the *may preorder* $K_1 \leq_{\text{may}} K_2$ between processes K_1 and K_2 holds iff the set of observers that K_1 *may satisfy* is contained in the set of observers that K_2 *may satisfy*. *May preorder* is equivalent to classical *trace inclusion*: when completely specified and deterministic systems are considered (e.g., classical FSMs), this is the only preorder of interest;
- the *must preorder* $K_1 \leq_{\text{must}} K_2$ between processes K_1 and K_2 holds iff the set of observers that K_1 *must satisfy* is contained in the set of observers K_2 *must satisfy*; checking for must preorder amounts to a comparison of the possible deadlock conditions for the two processes;
- according to [7], one can define a *testing preorder* $K_1 \leq_{\text{test}} K_2$ between processes K_1 and K_2, which holds iff $K_1 \leq_{\text{may}} K_2$ and $K_1 \leq_{\text{must}} K_2$; the testing preorder specifies that K_2 possesses all the traces of K_1 and does not present any deadlock condition which is not in K_1;
- as usual, equivalences can be defined from preorders: two systems are *must*-equivalent if the sets of observers they *must satisfy* is the same, they are *may*-equivalent if they *may satisfy* the same sets of observers, and they are *testing equivalent* if both sets coincide.

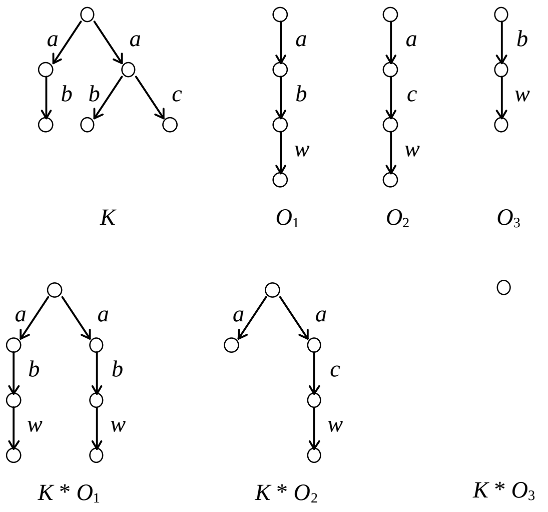

Figure 2: *Satisfaction of observers*

The above preorders play an important role in system description and verification methodologies, since they can describe the relation that holds between a specification and a correct implementation. Given a specification *Spec*:

- the set of computations that the implementation is requested to possess is modeled by the set of observers that *Spec* must satisfy;
- the set of computations that should never happen in a correct implementation is modeled by the observers that *Spec* can not satisfy.

During the design process (Fig. 3), one must generate an implementation *Impl* and guarantee that both the above sets of computations are not violated, while the observers that *Spec* may satisfy are a degree of freedom during design.

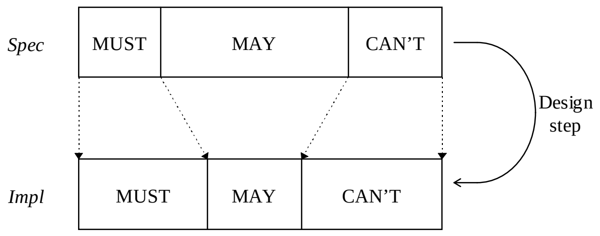

Figure 3: *Implementation Preorder*

This intuitive relation is called *implementation preorder*, and is defined as follows:

$$Spec \leq_{\text{impl}} Impl \text{ iff}$$
$$Spec \leq_{\text{must}} Impl \text{ and } Spec \geq_{\text{may}} Impl.$$

As the above definitions imply, checking for testing relations should involve the computation of all possible observers, which is not feasible in any practical realization. [8], to which the interested reader is referred, effectively bridges the gap between the abstract definition and an operational view. It provides an alternative characterization of the preorders which directly operates on LTSs, based on the definition of the set of actions that must be accepted *after* a specified sequence of actions has been executed.

3. Verification Algorithms

The algorithms implemented in SEVERO for the proof of testing preorders and equivalences are patterned after the conceptual procedure of [8] and are, to the best of our knowledge, the first successful attempt to implement testing relations with BDDs. The goal of this section is to introduce the reader to the strategy adopted for proofs and to give some details about the implemented symbolic algorithms.

Procedures for checking relations between transition systems have to solve two problems: the identification of the couples of states that are in correspondence (i.e., are reachable under the same I/O conditions) and the check of a given relation on such state couples. While the former is usually complex and involves traversals and fixed point computations on the state transition graph, the latter is a static property check. Usually, the two problems are solved together by means of some fixed point computation interleaved with the check of the property over each generated state set.

The implementation of testing relations we propose, instead, solves the two problems separately, in two different computation steps. This is convenient since different possible preorders exist and the result of the computation of the first step, by far more complex, can be reused several times.

The whole procedure can be schematically described as follows, assuming that one wants to check for some testing relation $\prec$ between processes K_1 and K_2:

- **Step 1**: transform the LTS of K_1 to a canonical form [8] by computing the smallest LTS K_1' still testing equivalent to K_1, and do the same for K_2. Minimization amounts to finding all the states that are reachable under the same conditions and which possess the same deadlock properties. The algorithm for minimization under the testing equivalence constraint is composed of two sub-steps:
  - **Step 1a**: build a deterministic finite automaton (DFA) D_1 starting from the LTS of K_1 by interpreting the LTS as a non deterministic finite automaton (NFA) and applying a BDD implementation of the classical conversion algorithm [13]. This sub-step also computes a mapping function $M_1(s_K, s_D)$ between each state s_D of the DFA D_1 and the set of states s_K corresponding to it in the LTS K_1. This information is needed in order to be able to detect, in the following sub-step, all the deadlock conditions.
  - **Step 1b**: convert the DFA D_1 to the canonical LTS K_1' by splitting each state according to the possible deadlock conditions it presents. This task is accomplished by identifying, through the mapping function $M_1(s_K, s_D)$, the set of states s_K corresponding to each state s_D and by partitioning them according to the set of possible actions. For each partition, a new state is generated in K_1'.
- **Step 2**: once the canonical forms K_1' and K_2' are available, check the desired property $\prec$ on them. This step has to check for *local* conditions only, since the minimal form is canonical and non determinism is confined to that introduced in Step 1b. This is again composed of two sub-steps:
  - **Step 2a**: compute a relation $R_{\prec}(s_1, s_2)$, called *compatibility relation*. This relation is the only part of the algorithm depending on the relation to be proven. It records which states s_1 of K_1' are in relation with s_2 in K_2'. The equations for the compatibility relations of the different preorders and equivalences are in Tab. 1, where $O(s, a, p)$ is true when a transition $s \xrightarrow{\tau} p \xrightarrow{a}$ exists.
  - **Step 2b**: check whether the possible combined evolutions of K_1 and K_2 are completely contained in $R_{\prec}(s_1, s_2)$, i.e., whether no state couple accessible to the systems violates the preorder or equivalence. This is done by computing the reachable state set R_{12} of the product machine of the two LTSs, and by checking for the inclusion $R_{12} \leq R_{\prec}$.

$R(s_1, s_2)$	**≥ Preorder**	**Equivalence**
May	$\neg\exists a\,(\neg\exists p\; O_1(s_1, a, p) \cdot \exists q\; O_2(s_2, a, q))$	$R_{\geq\text{may}}(s_1, s_2) \cdot R_{\leq\text{may}}(s_1, s_2)$
Must	$\forall p \exists q\,(\neg\exists a\,(O_2(s_2, a, q)\; \neg O_1(s_1, a, p)))$	$R_{\geq\text{must}}(s_1, s_2) \cdot R_{\leq\text{must}}(s_1, s_2)$
Test	$R_{\geq\text{must}}(s_1, s_2)$ $R_{\geq\text{may}}(s_1, s_2)$	$R_{\geq\text{test}}(s_1, s_2) \cdot R_{\leq\text{test}}(s_1, s_2)$
Implementation	$R_{\leq\text{must}}(s_1, s_2)$ $R_{\geq\text{may}}(s_1, s_2)$	$R_{=\text{test}}(s_1, s_2)$

Table 1: *Expressions for the Compatibility Relations*

4. Experimental results

This section presents an example to show the efficiency of the algorithms. To quantify the complexity of the proofs, a variant of the *dining philosophers* problem has been considered. The system is modeled by defining the behavior of each fork and philosopher as a separate process, and composing them. The fork models a shared resource, recording whether it is on the table or it is held by some philosopher. Two different configurations of the system were considered:

- **Case 1:** each philosopher gets his left fork, then the right one and releases them in the reverse order.
- **Case 2:** in order to avoid deadlock, philosophers are given the option of laying down their left fork whenever they are not able to acquire the right one.

The system has been chosen such that Case 2 can be an *implementation* of Case 1, i.e., *Case 1* $\leq_{\text{impl}}$ *Case 2*.

Tab. 2 and 3 summarize the results obtained with SEVERO on a SparcStation2 with 32 Mbytes of memory. A limit of 800,000 BDD nodes was set for all the computations and CPU times are reported in seconds. Column (a) reports the number n of philosophers and the system configuration. The size of the composite system, in terms of number of states and transitions, and the

CPU time to compose it are in columns (b), (c), and (d), respectively. Columns (e), (f) and (g) report results for obtaining the minimal LTS (Step 1), and (h) the time needed to estabilish the existence of the implementation preorder (Step 2).

5. Conclusions

This paper presented some sophisticated equivalence notions for Process Algebra descriptions and the corresponding proof algorithms. *Testing equivalences* and *testing preorders* are shown to be much more expressive and suitable for system-level descriptions than trace equivalence or bisimulations. In particular, the implementation preorder models the relation that must hold between a specification and one of its correct implementations.

For the first time, testing relations are implemented resorting to symbolic techniques and BDDs, and are implemented in the efficient tool SEVERO. Experimental results show that proofs of testing relations can be accomplished in acceptable time.

Acknowledgments

The authors wish to thank Prof. Rocco De Nicola for the useful discussions.

References

[1] A. Bouali, R. de Simone: *Symbolic Bisimulation Minimization*, CAV'92: 4th Workshop on Computer-Aided Verification, June 1992, pp. 97-108

[2] K.S. Brace, R.L. Rudell, R.E. Bryant: *Efficient Implementation of a BDD Package*, DAC'90: 27th ACM/IEEE Design Automation Conference, Orlando, FL (USA), June 1990, pp. 40-45

[3] R. E. Bryant: *Graph-based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, vol. C-35, N. 8, August 1986, pp. 677-691

[4] P. Camurati, F. Corno, P. Prinetto: *Exploiting symbolic traversal techniques for efficient Process Algebra Manipulation*, CHDL'93: IFIP Conference on Hardware Description Languages, Ottawa (CAN), pp. 21-34

[5] P. Camurati, F. Corno, P. Prinetto: *An efficient tool for system-level verification of behaviors and temporal properties*, EURO-DAC'93: IEEE European Design Automation Conference, Hamburg (D), September 1993, pp. 124-129

[6] R. Cleaveland, J. Parrow, B. Steffen: *The Concurrency Workbench*, "Automatic Verification Methods for Finite State Systems," J. Sifakis Editor, LNCS 407, Springer Verlag, Berlin (Germany), pp. 24-37

[7] R. De Nicola: *Extensional Equivalences for Transition Systems*, Acta Informatica, vol. 24, pp. 211-237, Springer-Verlag, New York, NY (USA), 1987

[8] R. De Nicola, M. Hennessy: *Testing Equivalences for Processes*, Theoretical Computer Science, vol.34, pp. 83-133, North Holland, Amsterdam (NL), 1984

[9] J. C. Godskesen, K. G. Larsen, M. Zeeberg: *TAV (Tools for Automatic Verification) Users Manual*, Dept. of Mathematics and Computer Science, Institute for Electronic Systems, Aalborg (DK), August 1989

[10] M. Hennessy, R. Milner: *Algebraic Laws for Nondeterminism and Concurrency*, J. ACM, 32 (1985), pp. 137-161

[11] M. Hennessy: *Algebraic Theory of Processes*, MIT Press, Cambridge, Mass. (USA), 1988

[12] C. A. R. Hoare, *Communicating Sequential Processes*, International Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ (USA), 1985

[13] J. E. Hopcrofr, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass, 1979

[14] R. Milner: *A Calculus of Communicating Systems*, Lecture Notes Computer Science, vol. 92, Springer-Verlag, New York, NY (USA), 1980

[15] G. J. Milne: *Circal and the Representation of Communication, Concurrency and Time*, ACM Transactions on Programming Languages and Systems, vol. 7, 1985

[16] R. Milner: *Communication and Concurrency*, Prentice Hall, Englewood Cliffs, NJ (USA), 1989

[17] G. J. Milne: *The Formal Description and Verification of Hardware Timing*, IEEE Trans. on Computers, 40, N. 7, July 1991, pp. 811-826

System	Build system		
n / case	#S	#T	Time [s]
5/1	392	5193	6.21
5/2	392	11254	6.76
6/1	1297	28890	20.34
6/2	1297	72890	27.53
7/1	4286	160317	70.66
7/2	4286	471454	83.63
(a)	(b)	(c)	(d)

Table 2: *Experimental results*

System	Mimimization			Preorder
n / case	#S	#T	Time [s]	Time [s]
5/1	17	56	31.66	
5/2	11	50	7.40	2.23
6/1	31	149	152.54	
6/2	18	136	17.83	6.45
7/1	51	330	843.58	
7/2	29	308	49.19	25.84
(a)	(e)	(f)	(g)	(h)

Table 3: *Experimental results (cont'd)*