Fundamenta Informaticae XXI (2008) 1–31
IOS Press

Testing Semantics for RPTA

Luis Llana and Manuel Núñez*

Dpto. Sistemas Informáticos y Computación.

Universidad Complutense de Madrid.

28040 Madrid. Spain.

e-mail: llana@sip.ucm.es, mn@sip.ucm.es

Abstract. The language RTPA, Real Time Process Algebra, has been created to enable rigorous treatment of knowledge representation and manipulation in terms of to be I to have / to do in a formal and coherent framework. This language has been designed to cope with the three dimensions involved in the problem of software specification: (i) mathematical operations, (ii) event/process timing, and (iii) memory manipulation. In this paper we focus on giving a testing semantics to the second dimension: Process timing dimension. First, we will provide a SOS like operational semantics for the process relations of RTPA. Next, we will define what a test is and we will introduce a relation based on which tests are passed by processes. Finally, we will obtain an operational characterization that can be used as a first step to define a denotational sematics sound and complete with respect the testing semantics.

Keywords: RTPA, real time process algebra, testing semantics

1. Introduction

During the last years, *cognitive informatics* [36, 38, 44, 42] is consolidating its position among more mature research schools. In fact, this area is emerging as a new, separate, interdisciplinary branch of

Address for correspondence: Luis Fernando Llana Díaz
Dpto. Sistemas Informáticos y Computación.
Universidad Complutense de Madrid.
Ciudad Universitaria. 28040 Madrid. Spain.
*Research partially supported by the Spanish MCYT project TIN2006-15578-C02-01, the Junta de Castilla-La Mancha project PAC06-0008, and the Marie Curie project MRTN-CT-2003-505121/TAROT.

No.	Meta-process	Syntax	Operational Semantics
1.1	System	$\S(SysID_S)$	Represents a system, $SysID$, identified by a string, S
1.2	Assignment	$y_{Type} := x_{Type}$	if $x.type = y.type$ then $x.value \Rightarrow y.value$ else $!(\text{@AssigmentTypeError}_S)$ where $Type \in \text{Meta} - \text{Types}$
1.3	Addressing	$ptrP\hat{} := x_{Type}$	if $prt.type = x.type$ then $x.value \Rightarrow ptr.value$ else $!(\text{@AssigmentTypeError}_S)$ where $Type \in \{H, Z, P\hat{}\}$
1.4	Input	$\text{Port}(ptrP\hat{})_{Type}\| > x_{Type}$	if $\text{Port}(ptrP\hat{}).type = x.type$ then $\text{Port}(ptrP\hat{}).value \Rightarrow x.value$ else $!(\text{@InputTypeError}_S)$ where $Type \in \{B, H\}$, $P\hat{} \in \{H, N, Z\}$
1.5	Output	$x_{Type}\| < \text{Port}(ptrP\hat{})_{Type}$	if $\text{Port}(ptrP\hat{}).type = x.type$ then $x.value \Rightarrow \text{Port}(ptrP\hat{}).value$ else $!(\text{@OutputTypeError}_S)$ where $Type \in \{B, H\}$, $P\hat{} \in \{H, N, Z\}$
1.6	Read	$\text{Mem}(ptrP\hat{})_{Type} > x_{Type}$	if $\text{Mem}(ptrP\hat{}).type = x.type$ then $\text{Mem}(ptrP\hat{}).value \Rightarrow x.value$ else $!(\text{@ReadTypeError}_S)$ where $Type \in \{B, H\}$, $P\hat{} \in \{H, N, Z\}$

Table 1. $RTPA$ meta-processes (1/3).

research. Even though there has been a lot of progress, cognitive informatics is still in a preliminary phase.Thus, we still need good formalisms to represent the different actors involved in the behavior of cognitive processes. In order to show that this is a truly interdisciplinary area, researchers in cognitive informatics very often take advantage of developments in other areas. Thus, they often adapt implementation and representation languages, theoretical models, and algorithms as well as consider best practices obtained in these areas. In this line, *descriptive mathematics for cognitive informatics* introduced as a representation language an adaption of classical process algebras [16, 24, 2, 25, 3]. Process algebras are very suitable to formally specify systems where concurrency is an essential key. This is so because they allow to model these systems in a compositional manner. Next, we briefly review the main milestones in the development of process algebras (see [3] for a good overview of the current research topics in the field). The first work on process algebras settled an important theoretical background for the study of concurrent and distributed systems. In fact, this work was very significant, mainly to shed light on concepts and to open research methodologies. However, due to the abstraction of the complicated features, models were still far from real systems. Therefore, some of the solutions were not specific enough, for instance, those related to real time systems. Thus, researchers in process algebras have tried to bridge the gap between formal models described by process algebras and real systems. In particular, features which were abstracted before have been introduced in the models. Thus, they allow the design of systems where not only functional requirements but also performance ones are included. The most significant of these additions are related to notions such as time (e.g. [34, 26, 45, 8, 17, 18, 1]) and probabilities (e.g.

No.	Meta-process	Syntax	Operational Semantics
1.7	Write	$x_{Type} < \mathrm{Mem}(ptrP\hat{})_{Type}$	if $\mathrm{Mem}(ptrP\hat{}).type = x.type$ then $x.value \Rightarrow \mathrm{Mem}(ptrP\hat{}).value$ else !(@WriteTypeError$_S$) where $Type \in \{B, H\}$, $P\hat{} \in \{H, N, Z\}$
1.8	Timing	a) $@t_{\mathtt{hh:mm:ss:ms}} := \S t_{\mathtt{hh:mm:ss:ms}}$ b) $@t_{\mathtt{yy:MM:dd}} := \S t_{\mathtt{yy:MM:dd}}$ c) $@t_{\mathtt{yy:MM:dd:hh:mm:ss:ms}} :=$ $\S t_{\mathtt{yy:MM:dd:hh:mm:ss:ms}}$	if $@t.type = \S t.type$ then $\S t.value \Rightarrow @t.value$ else !(@TimingTypeError$_S$) where $\mathtt{yy} \in \{0, \ldots, 99\}$, $\mathtt{MM} \in \{1, \ldots, 12\}$, $\mathtt{dd} \in \{1, \ldots, 31\}$, $\mathtt{hh} \in \{0, \ldots, 23\}$, $\mathtt{mm}, \mathtt{ss} \in \{0, \ldots, 59\}$, $\mathtt{ms} \in \{0, \ldots, 999\}$
1.9	Duration	$@tn_Z := \S tn_Z + \Delta n_Z$	if $@tn.type = \Delta n.type = \S tn.type = Z$ then $(\S tn.value + \Delta n.value)$ mod MaxValue $\Rightarrow @tn.value$ where MaxValue is the upper bound of the system relative-clock, and the unit of all values is ms
1.10	Memory allocation	AllocateObject (ObjectID$_S$, NofElements$_N$, ElementType$_{RT}$)	$n_N :=$ NofElements$\rightarrow$ $R^{n}_{i=1}$(new ObjectID(i_N) : ElementType$_{RT}$)$\rightarrow$ Ⓢ ObjectID.Existed$_{BL}$:= $\mathtt{true}$
1.11	Memory release	ReleaseObject(ObjectID$_S$)	delete ObjectID$_S$// System.Garbage Collection( ) $\rightarrow$ ObjectID$_S$:= null $\rightarrow$ Ⓢ ObjectID.Release$_{BL}$:= $\mathtt{true}$

Table 2. *RTPA* meta-processes (2/3).

[10, 29, 28, 7, 6, 27]). An attempt to integrate time and probabilistic information has been given by introducing *stochastic process algebras* (e.g. [11, 15, 4, 13, 5, 32, 19, 23]). The idea underlying the definition of stochastic time is that the actual amount of time is given by taking into account different probabilities.

It is worth to note that even though process algebras were not originally created to describe cognitive processes, they are a very suitable mechanism to do it since they are very appropriate to define systems where concurrency plays a fundamental role. In fact, variants of process algebras have been already used several times to represent *human* processes (see, for example, [30, 22, 39, 43, 31]). The first real *cognitive process algebra* is RTPA [37, 40]. conveniently putting together the ideas underlying the definition of classical process algebras, RTPA is a new mathematical framework to represent cognitive processes and systems. As professor Y. Wang states in [41]:

> *RTPA is a real-time process algebra that can be used to formally and precisely describe and specify architectures and behaviors of human and software systems.*

By Following RTPA, the process algebra STOPA [20, 33, 21] represents an advance since the notion of time is further developed to consider the representation of *stochastic time*.

The process algebra literature includes numerous semantic models. These semantic frameworks are used to describe the behavior of processes as well as to define relations on them. Testing semantics [9, 14] represents one of these semantic frameworks. Intuitively, two processes are *testing equivalent* if they

No.	Meta-process	Syntax	Operational Semantics
1.12	Increase	$\uparrow (n_{Type})$	if $n.value <$ MaxValue then $n.value + 1 \Rightarrow n.value$ else !(@ValueOutofRange$_S$) where $Type \in \{N, Z, B, H, P\hat{}\}$ MaxValue = min{run-time defined upper bound, nature upper bound of $Type$}
1.13	Decrease	$\downarrow (n_{Type})$	if $n.value > 0$ then $n.value - 1 \Rightarrow n.value$ else !(@ValueOutofRange$_S$) where $Type \in \{N, Z, B, H, P\hat{}\}$
1.14	Exception detection	!(@e_S)	$\uparrow$ (ExceptionLogPtr$_{P\hat{}}$)$\rightarrow$ @$e_S \Rightarrow$ Mem(ExceptionLogPtr$_{P\hat{}}$)$_S$
1.15	Skip	$\varnothing$	Exit a current control structure, such as loop, branch, or switch
1.16	Stop	stop	System stop

Table 3. $RTPA$ meta-processes (3/3).

have the same *responses* for all the *tests* belonging to a certain set. Depending on how these responses are analyzed, several testing semantics can be defined: may, must, refusal, fair, etc. We consider that this semantic framework is very suitable because it is easy to understand and allows us to give different equivalences just by modifying the idea of what a test is or when a test is successfully passed.

In this paper we adapt the classical testing framework to provide both a must and a may testing semantics for RTPA. The direct definition of these semantics relations is very hard to handle in the sense that we should apply an infinite number of test just to check if two processes are related. So, in addition to the definitions, we will give the corresponding characterizations based on the operational semantics of the processes.

The rest of the paper is structured as follows. In the next section we will recall from [37] the $RTPA$ language. This language is very expressive but too complex to formally study so, in Section 3, we present the core language that we will be the subject of our study. In Section 4 we present the SOS like operational semantics of the core language. Next, in Section 5 we present the definition of the *must* and *may* testing semantics. The characterization of both testing semantics will be given in terms of *states* and *barbs* that will be introduced in Section 6. Then, we present the characterization of the *must* testing semantics in Section 7 and the corresponding characterization of the *may* testing semantics in Section 8. Finally, in Section 9, we present the conclusions of our paper.

2. The $RTPA$ language

An RTPA *process* is a basic unit of software system behaviors that represents a transition procedure of a system from one state to another by changing its sets of inputs, outputs, and/or internal variables.

A process can be defined as a single *meta-process* or as a complex process based on meta-processes using *process relations*. Thus, $RTPA$ is described by using the following structure:

No.	Meta-type	Syntax
2.1	Natural number	N
2.2	Integer	Z
2.3	Real	R
2.4	String	S
2.5	Boolean	$BL = \{\mathtt{true}, \mathtt{false}\}$
2.6	Byte	B
2.7	Hexadecimal	H
2.8	Pointer	$P\hat{}$
2.9	Time	$\mathtt{hh} : \mathtt{mm} : \mathtt{ss} : \mathtt{ms}$ where $\mathtt{hh} \in \{0, \ldots, 23\}, \mathtt{mm}, \mathtt{ss} \in \{0, \ldots, 59\}$, $\mathtt{ms} \in \{0, \ldots, 999\}$
2.10	Date	$\mathtt{yy} : \mathtt{MM} : \mathtt{dd}$ where $\mathtt{yy} \in \{0, \ldots, 99\}, \mathtt{MM} \in \{1, \ldots, 12\}$, $\mathtt{dd} \in \{1, \ldots, 31\}$
2.11	Date/Time	$\mathtt{yyyy} : \mathtt{MM} : \mathtt{dd} : \mathtt{hh} : \mathtt{mm} : \mathtt{ss} : \mathtt{ms}$ where $\mathtt{yyyy} \in \{0, \ldots, 9999\}, \mathtt{MM} \in \{1, \ldots, 12\}$, $\mathtt{dd} \in \{1, \ldots, 31\}, \mathtt{hh} \in \{0, \ldots, 23\}$, $\mathtt{mm}, \mathtt{ss} \in \{0, \ldots, 59\}, \mathtt{ms} \in \{0, \ldots, 999\}$
2.12	Run-time determinable type	RT
2.13	System architectural type	ST
2.14	Event	$@e_S$
2.15	Status	Ⓢ s_{BL}

Table 4. $RTPA$ meta-types.

$RTPA$ ::= Meta-processes
| Primary types
| Abstract data types
| Process relations
| System architectures
| Specification refinement sequences

$RTPA$ is a formal method for specifying software system architectures, static and dynamic behaviors. This language distinguishes the concepts of meta-processes from complex processes and complex relations. A meta-process is an elementary process that serves as a basic building block in a software system. Complex processes can be derived from meta-processes according to given process combinatory rules. The syntax and operational semantics of the meta-processes are given in Tables 1, 2 and 3. Each meta-process is a basic operation on one or more operands such as variables, memory elements, or I/O ports. Structures of the operands and their allowable operations are contrained by their types.

The $RTPA$ notation is strongly typed. Every operand in $RTPA$ is assigned with a data type labeled as a suffix. The definition of the primary data types are the meta-types defined from 2.1 to 2.10 in Table 4. The meta-types date/time (2.11) are specialy types for continous real-time systems, where long-range timing manipulation is needed. The runtime determinable (2.12) is a subset of all the rest meta-types defined, which is designed to support flexible type specification that is unknown at compile-time, but will be instantiated at run-time. The system architectural components (2.13) is a novel and important

No.	ADT	Syntax	Designed behaviors
3.1	Stack	`Stack`:ST	Stack. (Create, Push, Pop, Clear, EmptyTest, FullTest, Release)
3.2	Record	`Record`:ST	Record. (Create, fieldUpdate, Update, FieldRetrieve, Retrieve, Release)
3.3	Array	`Array`:ST	Array. (Create, Enqueue, Serve, Clear, EmptyTest, FullTest, Release)
3.4	Queue (FIFO)	`Queue`:ST	Queue.(Create, Enqueue, Serve, Clear, EmptyTest, FullTest, Release)
3.5	Sequence	`Sequence`:ST	Sequence.(Create, Retrieve, Append, Clear, EmptyTest, FullTest, Release)
3.6	List	`List`:ST	List. (Create, FindNext, FindPrior, Findith, FindKey, Retrieve, Update, InsetAfter, InsertBefore, Delete, CurrentPos, FullTest, EmptyTest,SizeTest,Clear,Release)
3.7	Set	`Set`:ST	Set. (Create, Assign, In, Intersection, Union, Difference, Equal, Subset, Release)
3.8	File (Sequential)	`SeqFile`:ST	SeqFile. (Create, Reset, Read, Append, Clear, EndTest, Release)
3.9	File (Random)	`RandFile`:ST	RandFile. (Create, Reset, Read, Write, Clear, EndTest, Release)
3.10	Binary Tree	`BTree`:ST	BTree. (Create, TRaverse, Insert, DeleteSub, Update, Retrieve, Find, Characteristics, EmptyTest, Clear, Release)

Table 5. $RTPA$ abstract data types.

data type in $RTPA$ that models system architectural components. The event and status types are used to model systems event variables (2.14) as a string type, and system status variables (2.15) as a Boolean type.

In addition to the meta-types for system modeling, a set of typical and frequently used combinational data objects in system architectural modeling, the abstract data types are selected and predefined in $RTPA$. They are described in Table 5. The interested reader may find in [37] a detailed explanation of all the definitions appearing in Tables 1-5 as well as the definitions of *system architectures* and *specification refinement sequences*. In Tables 6 and 7 the syntax and operational semantics of the process relations are described.

3. The Core Language $RTPA_{core}$

The $RTPA$ language is a very high level language and that enormously complicates its formal study. Therefore we need to define a core language that we will call $RTPA_{core}$. This language will be slightly simpler than the process relations defined in $RTPA$, but it will be powerful enough to represent all process relations in $RTPA$.

In order to describe the $RTPA_{core}$ language, we will first introduce the set of actions Act. An action represents a change in the system: A communication of a value, the evaluation of an expression, an event

No.	Process relation	Syntax	Operational semantics
4.1	Sequence	$P \to Q$	$P \to Q$
4.2	Branch	$(?expBL = \texttt{true}) \to P$	if $expBL = \texttt{true}$
		$\mid (?expBL = \texttt{false}) \to Q$	then P
			else Q
4.3	Switch	$?expNUM =$	case $expNUM =$
		$0 \to P_0$	$0 : P_0$
		$1 \to P_1$	$1 : P_1$
		$\dots$	$\dots$
		$n-1 \to P_{n-1}$	$n-1 : P_{n-1}$
		$else \to \varnothing$	$else : exit$
4.4	For-do	$R^{n}_{i=1} P(i))$	for $i := 1$ to n
			do $P(i)$
4.5	Repeat	$R^{expBL \neq \texttt{true}}_{\geq 1} P$	repeat P
			until $expBL \neq \texttt{true}$
4.6	While-do	$R^{expBL \neq \texttt{true}}_{\geq 0} P$	while $expBL = \texttt{true}$
			do P
4.7	Function call	$P \hookrightarrow F$	$P' \to F \to P''$ where $P = P' \cup P''$
4.8	Recursion	P $\circlearrowleft$ P	$P' \to P \to P''$ where $P = P' \cup P''$
4.9	Parallel	$P \parallel Q$	MPSC (multi-processor single clock)
			internal parallel
4.10	Concurrence	$P \oint Q$	MPMC (multi-processor multi-clock)
			external parallel
4.11	Interleave	$P \mid\mid\mid Q$	SPSC (single processor single clock)
			internal virtual parallel
4.12	Pipeline	$P >> Q$	$P \to Q$ and $\{P_{outputs}\} \Rightarrow \{Q_{inputs}\}$

Table 6. $RTPA$ process relations (1/2)

triggered by some component, etc. We will also need two special actions. The first one, τ, represents an autonomous change in some part of the system. The second special action, $\surd$, represents the successful termination. Then, the $RTPA_{core}$ is defined by BNF expression presented in Table 8. Next we describe the operators appearing in the language.

3.1. Stop and exit processes

The stop process (`STOP`) and exit operator (`EXIT`) are the inductive basis of the language. The first one represent a *blocked* process, that is, it cannot execute any actions but it has not finished yet. The second one represents a process that has successfully terminated.

3.2. Sequence

The sequence operator[1] ($P \to Q$) is just the same as the one appearing in the original Table 6 of $RTPA$ process relations: After the successful termination of P, the process Q is executed.

[1]We will also call this operator *process prefix*.

No.	Process relation	Syntax	Operational semantics
4.13	Time-driven dispatch	$@ti_{\texttt{hh:mm:ss:ms}}$ $\Lsh P_i,\ i \in \{1,\ldots,n\}$	$@t1_{\texttt{hh:mm:ss:ms}} \Lsh P_1$ $\mid @t2_{\texttt{hh:mm:ss:ms}} \Lsh P_2$ $\ldots$ $\mid @tn_{\texttt{hh:mm:ss:ms}} \Lsh P_n$
4.14	Event-driven dispatch	$@ei_S \Lsh P_i,\ i \in \{1,\ldots,n\}$	$@e1_S \Lsh P_1$ $\mid @e2_S \Lsh P_2$ $\mid \ldots$ $\mid @en_S \Lsh P_n$
4.15	Interrupt	$P \parallel \odot(@e_S \nearrow Q \searrow \odot)$	$P \parallel$ system interrupt capture; if $@e_S$ captured = `true` then (record interrupt point $\odot$ and variables $\Lsh Q$ $\to$ recover interrupted variables $\to$ return to the interrupt point $\odot$ and continue P)
4.16	Jump	$P \to Q$	$P \to$ goto $Q \to Q$

Table 7. $RTPA$ process relations (2/2)

$$
\begin{aligned}
P ::=\ & \texttt{STOP} \mid \texttt{EXIT} \mid P \to Q \mid P_1 + P_2 \mid et \to P \mid \\
& @t \Lsh P \mid P \| \odot(A \nearrow Q \searrow \odot) \mid P \parallel_A Q \mid P \setminus A \mid X \mid X := Q
\end{aligned}
$$

where $e \in Act \cup \{\tau\}$, $t \in Time$, $A \subseteq Act$ and $X \in ID$.

Table 8. $RTPA_{core}$ language

3.3. Choice

This operator is one the most important operators in the $RTPA_{core}$ language. It is used to represent several $RTPA$ relations: Branch, switch, time-driven dispatch, and event-driven dispatch. The behavior of $P+Q$ is the following: Initially, all the behavior of P and Q is available in $P+Q$. However, when an action is performed, $P+Q$ behaves like P or Q depending on the component that performed the action.

Since switch and branch operators in $RTPA$ use the symbol $\mid$, we might also choose this symbol for the choice operator. In spite of that, we have preferred to choose the symbol $+$ to avoid confusion with the vertical bar that separates the terms in the BNF expression and to avoid confusion with the parallel operator.

Next will show how the $RTPA$ relations mentioned above can be embedded in $RTPA_{core}$ using the choice operator.

3.3.1. Branch

For any boolean expression $expBL$ we consider two events: $expBL_t$ that is enabled when the expression is evaluated to true; and $expBL_f$ that is enabled when it is evaluated to false. Therefore, the RTPA process relation

$$(?expBL = \mathtt{true}) \to P \mid (?expBL = \mathtt{false}) \to Q$$

will be equivalent to the following expression in $RTPA_{core}$

$$(expBL_t \to P) + (expBL_f \to Q)$$

3.3.2. Switch

Analogously to the previous case, for any expression $expNUM$ we consider the following events: For all $0 \leq i \leq n$ we have the event $expNUM_i$ that is enabled if expression $expNUM$ is evaluated to i. Hence, the process relation

$$\begin{array}{l} ?expNUM = \\ \quad 0 \to P_0 \\ \quad 1 \to P_1 \\ \quad \dots \\ \quad n-1 \to P_{n-1} \\ \quad n \to P_n \\ \quad else \to \varnothing \end{array}$$

corresponds to the following $RTPA_{core}$ expression

$$(expNUM_0 \to P_0) + (expNUM_1 \to P_1) + \cdots (expNUM_{n-1} \to P_{n-1}) + (expNUM_n \to P_n)$$

3.3.3. Time-driven dispatch

In the process relation $@t_i \Lsh P_i$ for $1 \leq t_i \leq n$, the i-th process P_i is triggered at the system time t_i. Therefore, it corresponds to the RTPA-core expresion

$$@t_1 \Lsh P_1 + @t_2 \Lsh P_2 + \cdots + @t_n \Lsh P_n$$

3.3.4. Event-driven dispatch

In this case, the i-th process P_i is triggered by a system event e_i. Thus, the $RTPA_{core}$ expression that represents the behavior of $@e_i \Lsh P_i \; i \in \{1, \dots, n\}$ is

$$e_1 \to P_1 + e_2 \to P_2 + \cdots + e_n \to P_n$$

3.4. Action prefix

The behavior of the *action prefix* $et \to P$ is simple. First, it can execute the action e at time t and then it behaves like P. Let us not that $e \in Act \cup \{\tau\}$. If e is a *visible action*, that is $e \in Act$, the action has an effect that is observable from the outside. On the contrary, if $e = \tau$ then the effect of the action cannot be directly observable.

3.5. Delay operator

The delay operator is used to add time constraints to the system. The process $@t{↳}P$ delays the execution of P until t time units pass (years, months, days, hours, minutes, seconds, milliseconds).

3.6. Interrupt Operator

The interrupt operator corresponds with the $RTPA$ interrupt operator. The process $P\|\odot(A \nearrow Q \searrow \odot)$ behaves like P, when an event $e \in A$ is triggered, then P is interrupted and Q is executed. Finally, when Q terminates, P is recovered in the state when the interruption happened.

3.7. Parallel Operator

Our parallel operator $P \parallel_A Q$ represents a parallel environment with the same clock where both parts synchronize in the set of actions A. If they synchronize in the set of all actions $(A = Act)$, both process execute the actions at the same time just like indicated in the original $RTPA$ semantics for process $P \parallel Q$ (4.9 in Table 6). On the contrary, if $A = \varnothing$ then the actions of P and Q are interleaved, just as indicated in the original $RTPA$ semantics for the process relation $P \mid\mid\mid Q$ (4.11 in Table 6).

3.8. Hiding operator

The hiding operator $P \setminus A$ hides the actions in the set A. These actions are considered internal. Thus, their effect cannot be seen directly from the outside.

3.9. Process identifier and Recursion

Recursion $(X := Q)$ is the only loop like operator that we have defined in $RTPA_{core}$. It is well known that any loop operator can be simulated with recursion and conditionals. Hence, we have preferred to leave only this operator in order to have a simpler syntax. In order to express recursion we need a set of *process identifiers* ID. Each process is defined by an expression that contains process identifiers.

It is important to remark that we will only consider *guarded recursion*, that is, any recursion call must be preceded by a prefix operator (action prefix or process prefix). Although it is difficult to explicit this condition in the BNF expression, it is easy to check if recursion is guarded for a given process. The reason to forbid non guarded recursion is that it would yield to undefined behaviors. Let us consider the following process

$$X := X$$

This process is allowed by the syntax but it is not clear what its behavior should be. Hence, to avoid pointless discussions about the meaning of the above process, we simply consider it a useless process.

Next we will see how the loop operators in $RTPA$ relations can be expressed in $RTPA_{core}$ by using recursion.

3.9.1. For-do

Let us suppose that we have the process relation $P = R_{i=1}^{n} P(i)$. This process relation can be translated to the following set of $RTPA_{core}$ processes

$$\begin{aligned} &P_0 := \texttt{EXIT} \\ &P_i := P(i) \rightarrow P_{i-i},\ 1 \leq i \leq n \\ &P := P_n \end{aligned}$$

3.9.2. Repeat

Now let us consider the process relation $P = R_{\geq 1}^{expBL \neq \texttt{true}} P'$. The process in $RTPA_{core}$ corresponding to the previous process relation is the following

$$P := P' \rightarrow (expBL_f \rightarrow \texttt{EXIT} + expBL_t \rightarrow P' \rightarrow P)$$

3.9.3. While-do

In this case we have $P = R_{\geq 0}^{expBL \neq \texttt{true}} P'$. That can be encoded as

$$P := expBL_f \rightarrow \texttt{EXIT} + expBL_t \rightarrow P' \rightarrow P$$

4. Operational Semantics

In order to give a testing semantics following [14], first we have to provide the language with an SOS operational semantics that is presented in Table 9. The rules of the operational semantics need an auxiliary function $\mathsf{upd}(t, P)$, which represents the passing of t time units on P. Formally, it is a function

$$\mathsf{upd}(\cdot,\cdot) : Time \times RTPA_{core} \longmapsto RTPA_{core}$$

that is inductively defined in Table 10. It is important to note that in the definition of this function we have omitted the case of recursion. This is because the recursive calls of the $\mathsf{upd}(\cdot,\cdot)$ function do not propagate to the right term of the prefix operators (action prefix and process prefix). Thus, since we are only considering guarded recursion, it is not necessary to explicit the value of the function in the case of recursion.

Formally, the operational semantics of $RTPA_{core}$ is given by the relation

$$\dot{\rightarrow} \subseteq RTPA_{core} \times (Act \cup \{\tau, \surd\} \times Time) \times RTPA_{core}$$

defined by the rules in Table 9. First, let us give a brief explanation of the rules.

$[\mathbf{EXIT}]$ The only action that EXIT process can execute is successful termination. Successful termination is executed as soon as it is available.

$[\mathbf{PRE}]$ This rule specifies that the process $et \rightarrow P$ can execute action e at time t.

$$[\mathbf{EXIT}] \quad \mathtt{EXIT} \xrightarrow{\surd 0} \mathtt{STOP}$$

$$[\mathbf{PRE}] \quad et \rightarrow P \xrightarrow{et} P, \quad e \in Act \cup \{\tau\}$$

$$[\mathbf{SEQ1}] \quad \frac{P \xrightarrow{et} P', \quad P \overset{\surd t'}{\not\longrightarrow}\ t' < t}{P \rightarrow Q \xrightarrow{et} P' \rightarrow Q}, \qquad e \neq \surd$$

$$[\mathbf{SEQ2}] \quad \frac{P \xrightarrow{\surd t} P', \quad P \overset{\surd t'}{\not\longrightarrow}\ t' < t}{P \rightarrow Q \xrightarrow{it} Q}$$

$$[\mathbf{CH1}] \quad \frac{P \xrightarrow{at} P', \qquad Q \overset{it'}{\not\longrightarrow}\ t' < t}{P + Q \xrightarrow{at} P'} \qquad\qquad \frac{P \xrightarrow{at} P', \qquad Q \overset{it'}{\not\longrightarrow}\ t' < t}{Q + P \xrightarrow{at} P'}$$

$$[\mathbf{INT1}] \quad \frac{P \xrightarrow{et_1} P'}{P \| \bigodot (E \nearrow Q \searrow \bigodot) \xrightarrow{at_1} P'}, \quad e \notin E$$

$$[\mathbf{INT2}] \quad \frac{P \xrightarrow{et_1} P'}{P \| \bigodot (E \nearrow Q \searrow \bigodot) \xrightarrow{\tau t_1} Q \rightarrow (P' \| \bigodot (E \nearrow Q \searrow \bigodot))}, \quad e \in E$$

$$[\mathbf{DEL}] \quad \frac{P \xrightarrow{et_1} P'}{@t\text{↳}P \xrightarrow{e(t+t_1)} P'}$$

$$[\mathbf{PAR}] \quad \frac{P \xrightarrow{et} P', \quad Q \overset{it'}{\not\longrightarrow}\ t' < t}{P \parallel_A Q \xrightarrow{et} P' \parallel_A \mathsf{upd}(t, Q)}, \quad e \notin A \qquad\qquad \frac{P \xrightarrow{et} P', \quad Q \overset{it'}{\not\longrightarrow}\ t' < t}{Q \parallel_A P \xrightarrow{et} \mathsf{upd}(t, Q) \parallel_A P'}, \quad e \notin A$$

$$[\mathbf{SYN}] \quad \frac{P \xrightarrow{at} P', \quad Q \xrightarrow{at} Q'}{P \parallel_A Q \xrightarrow{at} P' \parallel_A Q'}, \quad a \in A$$

$$[\mathbf{HD1}] \quad \frac{P \xrightarrow{et} P', \quad P \overset{at'}{\not\longrightarrow}\ t' < t,\ a \in A}{P \setminus A \xrightarrow{et} P' \setminus A}, \quad e \notin A$$

$$[\mathbf{HD2}] \quad \frac{P \xrightarrow{at} P', \quad P \overset{a't'}{\not\longrightarrow}\ t' < t,\ a' \in A}{P \setminus A \xrightarrow{it} P' \setminus A}, \quad a \in A$$

$$[\mathbf{REC}] \quad \frac{X := P,\ P \xrightarrow{et} P'}{X \xrightarrow{et} P'}$$

Table 9. Operational Semantics.

$[\mathbf{SEQ1}]$ **and** $[\mathbf{SEQ2}]$ These rules indicates the behavior of the sequence operator. Its behavior is the one indicated by the original $RTPA$ semantics defined in Table 6. The process $P \rightarrow Q$ behaves like the first component P until it successfully terminates; then, it behaves like the second component Q.

$[\mathbf{CH1}]$ The behavior of $P + Q$ is specified by this rule. Initially, the process $P + Q$ is able to execute the actions that are available to either P or Q. Whenever an action is executed, the process behaves like the process that has executed the action.

$[\mathbf{INT1}]$ **and** $[\mathbf{INT2}]$ The interrupt operator behaves as indicated in the original $RTPA$ semantics defined in Table 7. The process $P \| \bigodot (A \nearrow Q \searrow \bigodot)$ initially behaves like P. If a captured event is triggered ($a \in A$) then the execution of P is suspended and Q is executed. When Q successfully terminates, the process P is resumed in the state it was when the event was triggered.

$[\mathbf{DEL1}]$ This rule specifies the delay operator. The process $@t\text{↳}P$ can perform the same actions as the

$$\mathsf{upd}(t,P) = \begin{cases} P & \text{if } P = \mathtt{STOP},\ P = \mathtt{DIV},\ \text{or} P = \mathtt{EXIT}, \\ \mathtt{STOP} & \text{if } P = et_1 \rightarrow P_1 \text{and } t_1 < t \\ e(t_1 - t) \rightarrow P_1 & \text{if } P = et_1 \rightarrow P_1 \text{and } t_1 \geq t \\ \mathsf{upd}(t, P_1) \rightarrow P_2 & \text{if } P = P_1 \rightarrow P_2 \\ @(t_1 - t)\Lsh P & \text{if } P = @t_1 \Lsh P \text{and } t_1 > t \\ \mathsf{upd}(t - t_1, P) & \text{if } P = @t_1 \Lsh P \text{and } t_1 \leq t \\ \mathsf{upd}(t, P_1)\ op\ \mathsf{upd}(t, P_2) & \text{if } P = P_1\ op\ P_2,\ op \in \{+, \|_A\} \\ \mathsf{upd}(t, P_1) \| \bigodot(A \nearrow P_2 \searrow \bigodot) & \text{if } P = P_1 \| \bigodot(A \nearrow P_2 \searrow \bigodot) \\ \mathsf{upd}(t, P_1) \setminus A & \text{if } P = P_1 \setminus A \end{cases}$$

Table 10. Auxiliary function $\mathsf{upd}(t, P)$.

process P but delayed t time units.

$[\mathbf{PAR}]$ **and** $[\mathbf{SYN}]$ These rules specify the behavior of the parallel composition $P \|_A Q$. This process interleaves the actions that both processes (P and Q) can execute as long as they do not belong to the set of synchronization events A (Rule $[\mathbf{PAR}]$). If the action belongs to A then both processes must execute it synchronously (Rule $[\mathbf{SYN}]$).

$[\mathbf{HD1}]$ **and** $[\mathbf{HD2}]$ The hiding operator hides the actions belonging to the set A. So, the process $P \setminus A$ behaves like P as long as it does not execute actions from the set A. This operator does not prevent the execution of actions in the set A; it just consider them internal actions. Internal actions do not need any interaction. So, there is no reason to delay its execution and hence, they are executed as soon as they are available. That is why we have to check that no actions from the set A may be executed before any other action is executed.

$[\mathbf{REC}]$ Finally, the recursion rule indicates that a process identifier behaves like the process that defines it.

Before going any further, we must point out that some rules in Table 9 have negative premises. This can be fatal for a SOS transition system because it could lead to contradictory assessments. To prevent this we follow [12], and we provide a *stratification* that guaranties the correctness of our transition system. The needed stratification is

$$f(P \xrightarrow{et} P') = t$$

This function is indeed a stratification since time values in $RTPA$, as indicated in Table 4, are specified in milliseconds $[\mathtt{yyyy} : \mathtt{MM} : \mathtt{dd} :]\mathtt{hh} : \mathtt{mm} : \mathtt{ss} : \mathtt{ms}$.

The following two Lemmas state two important properties of the operational semantics. Both are easily proved by structural induction; the proofs are quite straight forward but cumbersome, so we omit the details. The first property is usually called *urgency*, that is, internal actions are executed as soon as they are available. Nevertheless, internal actions have not greater priority than observable actions to be executed at the same time. Finally, observable actions to be executed before some internal action are not affected by the existence of such an internal action.

Lemma 4.1. If $P \xrightarrow{it} P'$ then $P \overset{et'}{\not\longrightarrow}$ for any $e \in Act \cup \{\tau, \surd\}$ and $t' > t$. □

The second important property is that this operational semantics is *finite branching*, that is, *for a given process P there exists a finite number of transitions that P can execute*.

Lemma 4.2. Let $e \in Act \cup \{\tau, \surd\}$ and $t \in Time$. Then set $\{P' \mid P \xrightarrow{et} P'\}$ is finite. □

Finally, let us comment that is quite easy to define abnormal behaviors in the defined framework. We are referring to Zeno-like processes: Those that can perform an infinite number of actions in a finite amount of time.

Example 4.1. Let us consider the following $RTPA_{core}$ process

$$\texttt{DIV} := \tau 0 \rightarrow \texttt{DIV}$$

This process has the following computation, all the actions being performed at time 0:

$$\texttt{DIV} \xrightarrow{\tau 0} \texttt{DIV} \xrightarrow{\tau 0} \texttt{DIV} \cdots$$

This behavior is obviously impossible so it must be avoided. □

5. Testing Semantics

Once we have defined a SOS operational semantics for $RTPA_{core}$, we can define a testing semantics. In order to identify the abnormal behavior in Example 4.1, we introduce the *convergence* predicate. Just as a step to define it, we need the *weak* predicate

Definition 5.1. We define the *weak convergence* predicate over $RTPA_{core}$, denoted by $P\downarrow$, as the least predicate that satisfies:

- $\texttt{STOP}\downarrow \texttt{EXIT}\downarrow$ and $a \rightarrow P\downarrow$, for $a \in Act$.
- If $t > 0$ or $P\downarrow$ then $@t\Lsh P\downarrow$.
- If $P\downarrow$ and $Q\downarrow$ then $P\ op\ Q\downarrow$, $op \in \{\|_A, +\}$.
- If $P\downarrow$ then $P \rightarrow Q\downarrow$, $P \setminus A\downarrow$ and $X\downarrow$ (where $X := P$).

We define the *strong convergence* predicate, or simply *convergence*, over $RTPA_{core}$, denoted by $P\Downarrow$, as the least predicate that satisfies

$$P\Downarrow \quad \text{iff} \quad P\downarrow \text{ and } \forall P' (P \xrightarrow{\tau 0} P' \text{ implies } P'\Downarrow).$$

We say that a process diverges, which is denoted by $P\Uparrow$, if $P\Downarrow$ does not hold. If $t \in Time$, it is useful to define the predicate $P\Downarrow_t$ as

$$P\Downarrow_t \quad \text{iff} \quad P\Downarrow \text{ and } \forall P' \forall t' \leq t\, (P \xrightarrow{\tau t'} P' \text{ implies } P'\Downarrow).$$

□

$$T ::= \ \texttt{OK} \mid \texttt{STOP} \mid \texttt{EXIT} \mid T_1 \to T_2 \mid T_1 + T_2 \mid et \to T \mid$$
$$@t\lrcorner T \mid T_1 \| \bigodot (A \nearrow T_2 \searrow \bigodot) \mid T_1 \parallel_A T_2 \mid T \setminus A \mid X \mid X := T$$

where $e \in Act \cup \{\tau\}$, $t \in Time$, $A \subseteq Act$ and $X \in ID$.

Table 11. Tests

Let us note that a process only diverges due to an infinite computation of internal actions to be instantaneously executed, that is, at time 0.

Tests are very similar to processes but defined by considering an extended grammar where we add a new process, `OK`, which expresses that the test has been successfully passed. More precisely, tests are generated by the BNF expression given in Table 11.

Now, with test we will use a notation we have not used so far. If $A = \{e_1t_1, e_2t_2, \cdots, e_nt_n\}$ is a finite set such that $e_i \in Act \cup \{\tau\}$, $t_i \in Time$, and for all $et \in A$ there is a test T_{et}, we denote the test

$$(e_1t_1 \to T_{e_1t_1}) + (e_2t_2 \to T_{e_2t_2}) + \cdots + (e_nt_n \to T_{e_nt_n})$$

$$\text{by} \qquad \sum_{et \in A} et \to T_{et}$$

The operational semantics for tests is defined in the same way as for plain processes, but only including a rule for the test `OK`[2]:

$$[\mathbf{OK}] \quad \texttt{OK} \xrightarrow{\texttt{OK}} \texttt{STOP}.$$

Finally, we define the composition of a test and a process as

$$P \mid T = (P \parallel_{Act} T) \setminus Act$$

That is, the process and the test must synchronize in all visible actions, being this actions automatically hidden.

Now we can relate processes and tests. We execute a process in parallel with a test and we study the computations that they produce. In particular, we are interested in those computations where the tests can execute the `OK` action. A computation is successful if the test can execute the `OK` action.

Definition 5.2. Let P be a $RTPA_{core}$ process and T be a test. Given a computation of $P \mid T$

$$P \mid T = P_1 \mid T_1 \xrightarrow{\tau t_1} P_2 \mid T_2 \cdots P_k \mid T_k \xrightarrow{\tau t_k} P_{k+1} \mid T_{k+1} \cdots$$

we say that it is

- *Complete* if it is finite and blocked (no further step is allowed) or infinite.

[2]To be exact, we should extend the definition of the operational semantics of both processes and tests to mixed terms defining their composition, since these mixed terms are neither processes nor tests. Since this extension is immediate we have preferred to avoid this formal definition.

- *Successful* if there exists some k such that $T_k \xrightarrow{\texttt{OK}}$.

$\square$

Next we present the notion of test passing. Depending if all the computations of the parallel composition of a process and a test are successful, or just some computations are successful; we say that the process *must pass* or *may pass* the test.

Definition 5.3. Let P be a $RTPA_{core}$ process and T be a test.

- We say that P *must pass* the test T, denoted by P must T, iff all complete computations of $P \mid T$ are successful.
- We say that P *may pass* the test T, denoted by P may T, iff there exists at least one successful computation of $P \mid T$.

$\square$

Finally, we can define relations between processes. Intuitively, a process P is better than Q if it P passes more tests than Q. Since tests can be passed in *may sense* or in *must sense* we can define two relations:

Definition 5.4. Let P and Q be $RTPA_{core}$ processes.

- We write $P \sqsubseteq\!\!\!\sim_{\text{must}} Q$ iff whenever P must T we have also Q must T.
- We write $P \sqsubseteq\!\!\!\sim_{\text{may}} Q$ iff whenever P may T we have also Q may T.

$\square$

6. Operational characterization

Even though our testing semantics is very intuitive, it is hard to handle. In order to check whether $P \sqsubseteq\!\!\!\sim_{\text{must}} Q$ or $P \sqsubseteq\!\!\!\sim_{\text{may}} Q$, according to Definition 5.4, we should prove the property for all possible tests. Unfortunately, the number of tests is infinite. Therefore, it is necessary to find an alternative way to check the relationship between two processes. This alternative definition will be given directly from the operational semantics defined in Section 4. From that operational semantics, and independently from the tests, we can compute what we call *states* and *barbs* of a process.

6.1. Sets of states

In order to characterize our testing semantics we will consider some kind of sets of timed actions, which we call *states*, that represent any of the possible *local configurations* of a process. In a state we have the set of timed actions offered and we also have the time, if any, at which the process becomes undefined, that is, divergent. In order to capture divergence (or equivalently undefinition), with a simple notation, we introduce a new element $\Omega \notin Act$, that represents an undefined substate. Then, we consider the sets $Act_\Omega = Act \cup \{\Omega\}$, $TAct = Act \times Time$, and $Act_\Omega \times Time = Act_\Omega \times Time$.

Definition 6.1. We say that $A \subseteq Act_\Omega \times Time$ is a *state* if

- At most there is a single element $\Omega t \in A$, that is, $\Omega t,\ \Omega t' \in A$ implies $t = t'$.
- If $\Omega t \in A$ then t is the maximum time in A, that is, $\Omega t,\ at' \in A$ implies $t' < t$.

We will denote by $\mathcal{ST}$ the set of states. □

The following example illustrates the formerly introduced concept

Example 6.1. Let us consider the following $RTPA$ process

$$\begin{array}{lllll} P = & @00:00:00:01 \lfloor\!\!\!\rightarrow a \rightarrow \texttt{STOP} & & & \\ & @00:00:00:02 \lfloor\!\!\!\rightarrow \tau \rightarrow & @00:00:00:01 \lfloor\!\!\!\rightarrow b \rightarrow \texttt{STOP} & & \\ & & @00:00:00:01 \lfloor\!\!\!\rightarrow \tau \rightarrow & c \rightarrow \texttt{STOP}| & \\ & & & @00:00:00:01 \lfloor\!\!\!\rightarrow \tau \rightarrow \texttt{DIV} & \end{array}$$

This process is represented in $RTPA_{core}$ as the following process

$$P = (@1 \lfloor\!\!\!\rightarrow a0 \rightarrow \texttt{STOP}) + @2 \lfloor\!\!\!\rightarrow \tau 0 \rightarrow \left(\begin{array}{l} (@1 \lfloor\!\!\!\rightarrow b0 \rightarrow \texttt{STOP}) + \\ @1 \lfloor\!\!\!\rightarrow \tau 0 \rightarrow (c0 \rightarrow \texttt{STOP} + @1 \lfloor\!\!\!\rightarrow \tau 0 \rightarrow \texttt{DIV}) \end{array} \right)$$

In this process, if no action is executed and 4 units time pass, the process becomes DIV (see Example 4.1), that is undefined. If the surrounding processes want to synchronize in action a at time 1, this synchronization will take place for sure. The same happens with action c at time 3. But action b at time 3 if offered simultaneously with an internal action. So, if the surrounding processes want to synchronize with b at time 3, this synchronization may take place or not. These ideas can be formally expressed by saying that P has two states: $\{a1, c3, \Omega 4\}$ and $\{a1, b3, c3, \Omega 4\}$.

More generally, when we have a process such that $P \xrightarrow{it} P'$ and A is a state of P', the state A splits into two states: the state $S(P) \cup (A + t)$, in which actions at time t are included, and another state $(S(P) \upharpoonright t)^3 \cup (A + t)$, in which actions at time t are not. So, actions in $S(P)$ whose time is strictly less than t cannot be rejected, but actions just at time t could be rejected. □

Next we give some auxiliary definitions.

Definition 6.2.

- We define the function $\mathsf{nd}(\cdot) : \mathcal{ST} \mapsto Time \cup \{\infty\}$, which give us the time at which a state becomes undefined, (<u>n</u>ot <u>d</u>efined function), by:

$$\mathsf{nd}(A) = \begin{cases} t & \text{if} \Omega t \in A \\ \infty & \text{otherwise} \end{cases}$$

- Given a state $A \in \mathcal{ST}$ and a time $t \in Time$, we define:

$$A + t = \{a(t + t') \,|\, at' \in A\} \qquad A \upharpoonright t = \{at' \,|\, at' \in A \text{ and } t' < t\}$$

- If $A \in \mathcal{ST}$, we define its set of timed actions as $TAct(A) = A \upharpoonright \mathsf{nd}(A)$.
- If $A \subseteq TAct$ and $t \in Time$, we will say that $A < t$ (resp. $A \leq t$) iff for all $at' \in A$ we have $t' < t$ (resp. $t' \leq t$).

□

Finally, in order to define the set of states of a process, we have to define the partial order $\prec$ between states.

Definition 6.3. Given states A_1 and A_2, we write $A_1 \prec A_2$ iff $\mathsf{nd}(A_1) \leq \mathsf{nd}(A_2)$ and $TAct(A_1) = A_2 \upharpoonright \mathsf{nd}(A_1)$. □

The above definition can be read as follows: A state A_1 is *less defined* than a state A_2 if the following two conditions hold A_1 becomes undefined earlier than A_2 and before the time in which A_1 becomes undefined, both states include the same timed actions. It is easy to check that the relation $\prec$ is a partial order; moreover it is a complete partial order.

Definition 6.4. Given a nondecreasing chain of states $A_1 \prec A_2 \cdots$, we define

$$\mathsf{lub}(\{A_k \mid k \in \mathbb{N}\}) = \left(\bigcup_{i \in \mathbb{N}} TAct(A_i)\right) \cup \begin{cases} \{\Omega t\} & \text{if} \exists k\, \forall l \geq k\; \Omega t \in A_l \\ \varnothing & otherwise. \end{cases}$$

□

It is easy to check that the state $\mathsf{lub}(\{A_k \mid k \in \mathbb{N}\})$ defined above is the least upper bound of the nondecreasing chain $\{A_k \mid k \in \mathbb{N}\}$.

6.2. States of a Process

In order to define the set of states of a process we have first to define the initial set of timed actions that a process P can perform, which is given by

$$S(P) = \big\{at \mid P \xrightarrow{at},\ a \in Act\big\}.$$

We will generalize the procedure given in example 6.1. First, we observe that a state includes the actions that a process can execute after a *finite* number of internal actions. Here we find a very important difference with respect to untimed process algebras: A process P can execute an infinite sequence of internal actions without diverging, for instance $P := @1 \mathrel{\Lsh} \tau 0 \rightarrow P$. This is why we have first to define the set of states that can be reached after k steps.

Definition 6.5. Let P be a process and $k \in \mathbb{N}$. We define the set $\mathsf{st}(k, P)$ as the least set of states that satisfies:

- If $P \Uparrow$ then $\mathsf{st}(k, P) = \Big\{\{\Omega 0\}\Big\}$.
- If $P \Downarrow$ then

– If $P \overset{it}{\not\longrightarrow}$ then $\mathsf{st}(k,P) = \Big\{S(P)\Big\}$.

– If $P \xrightarrow{it} P'$ and $P' \Uparrow$, or $k = 1$, then $\mathsf{st}(k,P) = \Big\{(S(P) \upharpoonright t) \cup \{\Omega t\}\Big\}$.

– If $P \xrightarrow{it} P'$, $P \Downarrow_t$, $A \in \mathsf{st}(k-1,P')$ and $k > 1$, then $(S(P) \upharpoonright t) \cup (A+t),\ S(P) \cup (A+t) \in \mathsf{st}(k,P)$.

We say that $A \in \mathcal{ST}$ is a state of P, denoted by $A \in \mathcal{A}(P)$, iff there exists a sequence $\{A_i \in \mathsf{st}(i,P) \mid i \in \mathbb{N}\}$ such that $A_i \prec A_{i+1}$ and $A = \mathsf{lub}(\{A_i \mid i \in \mathbb{N}\})$. □

It is clear that the previous definition will be hard to use. So, an alternative characterization is necessary. The following proposition states how a state can be computed from a given computation of internal actions.

Proposition 6.1. Let P be a process. The set $\mathcal{A}(P)$ is the set of states $A \in \mathcal{ST}$ which can be generated as described below from a *complete* computation (what means that it is either infinite, or there exists a final process such that either no internal action is possible or it diverges)

$$P = P_1 \xrightarrow{\tau t_1} P_2 \xrightarrow{\tau t_2} \cdots P_k \xrightarrow{\tau t_k} P_{k+1} \cdots$$

where we denote by P_n the final process if the computation is finite, and taking $t^i = \sum_{j=1}^{i-1} t_j$, we have $P_i \Downarrow_{t^i}$ for all $i < n$.

- For each infinite computation we have $\bigcup_{i \in \mathbb{N}} (A_i + t^i) \in \mathcal{A}(P)$, where for each $i \in \mathbb{N}$ we have either $A_i = S(P_i)$ or $A_i = S(P_i) \upharpoonright t_i$.
- For each finite computation we have $\bigcup_{i=1}^{n} (A_i + t^i) \in \mathcal{A}(P)$, where for each $i \le n-1$ we have either $A_i = S(P_i)$ or $A_i = S(P_i) \upharpoonright t_i$; except if $P_n \Uparrow$ where A_{n-1} must be equal to $S(P_{n-1}) \upharpoonright t_{n-1}$; $A_n = S(P_n)$ if $P_n \Downarrow$, otherwise $A_n = \{\Omega 0\}$. □

Proof:
It is a consequence of the finite branching property of the operational semantics. □

6.3. Barbs

A *barb* is a generalization of an acceptance [14], but additional care must be taken about the actions that the process offers *before* any action has been executed. First we introduce the concept of *b-trace*, which is a generalization of the notion of trace. A b-trace, bs, is a sequence, $A_1 a_1 t_1 A_2 a_2 t_2 \cdots A_n a_n t_n$, that represents the execution of the sequence of timed actions $a_1 t_1 a_2 t_2 \cdots a_n t_n$, and after the execution of $a_1 t_1 \cdots a_{i-1} t_{i-1}$, the timed actions in A_i were offered before accepting $a_i t_i$. Then a barb is a b-trace followed by a state, that represents a configuration of a process after executing a b-trace.

To illustrate this let us recall Example 6.1. Where we have that process P may execute action b at time 3, but action a at time 1 is always possible. So, we have that process P can execute the b-trace $\{a1\}b3$ to become `STOP` afterwards. Urgency is the reason why we have to keep the information about the actions the process could have executed *before* any other. In this example, if the environment could synchronize in action a at time 1, the process should do it, and then action b at time 3 would be no longer possible. The full set of barbs of P is the following

$$\mathsf{Barb}(P) = \left\{ \begin{array}{l} \{a1, c3, \Omega 4\}, \{a1, b3, c3, \Omega 4\}, \\ \varnothing a1\varnothing, \{a1\}b3\varnothing, \{a1\}c3\varnothing \end{array} \right\}$$

The next definition formally introduce these concepts.

Definition 6.6.

- We say that a *b-trace* is a finite sequence, $bs = A_1a_1t_1 \cdots A_na_nt_n$, where $n \geq 0$, $a_it_i \in TAct$, $A_i \subseteq TAct$, and $A_i < t_i$. We say that $\mathsf{lon}(b) = n$; if $n = 0$ we have the empty b-trace denoted by ϵ.
- A *barb* b is a sequence $b = bs \cdot A$ where bs is a b-trace and A is a state. We will write the barb $\epsilon \cdot A$ as A, so we will consider any state A as a barb. □

In order to define the barbs of a process we need first the some auxiliary notations.

Definition 6.7. Given $t \in T$, a b-trace $bs = A_1a_1t_1 \cdot bs_1$ and a set of timed actions $A \subseteq TAct$ such that $A < t$, we define $(A, t) \sqcup bs = (A \cup (A_1 + t))a(t_1 + t) \cdot bs'$.

Let $P,\ P'$ be processes and bs be a b-trace. We define the relation $P \stackrel{bs}{\Longrightarrow} P'$ as follows

- $P \stackrel{\epsilon}{\Longrightarrow} P$.
- If $P \Downarrow_t$, $P \xrightarrow{it} P_1$, and $P_1 \stackrel{bs'}{\Longrightarrow} P'$ with $bs \neq \epsilon$ then $P \xRightarrow{(S(P)\upharpoonright t,t)\sqcup bs'} P'$.
- If $P \Downarrow_t$, $P \xrightarrow{at} P_1$, and $P_1 \stackrel{bs'}{\Longrightarrow} P'$ then $P \xRightarrow{(S(P)\upharpoonright t)at\cdot bs'} P'$.

Let $b = bs \cdot A$ be a barb and P be a process. We say that b is a barb of P, denoted by $b \in \mathsf{Barb}(P)$, iff there exists a process P' such that $P \stackrel{bs}{\Longrightarrow} P'$ and $A \in \mathcal{A}(P')$. □

7. Must Testing Semantics

We will use barbs to characterize the testing semantics, this will be done by defining a pre-order between sets of barbs. In order to define this pre-order, first we need the following order relations.

Definition 7.1.

- We define the relation $\ll$ between b-traces as the least relation that satisfies: 1. $\epsilon \ll \epsilon$, 2. If $bs' \ll bs$ and $A' \subseteq A$ then $A'at \cdot bs' \ll Aat \cdot bs$.
- We define the relation $\ll$ between barbs as the least relation that satisfies:
    1. If $bs,\ bs'$ are b-traces such that $bs' \ll bs$ and $A,\ A'$ are states such that $\mathsf{nd}(A') \leq \mathsf{nd}(A)$ and $TAct(A') \subseteq A$, then $bs' \cdot A' \ll bs \cdot A$.

2. If A' is a state, $b = A_1 a_1 t_1 \cdot b'$ is a barb such that $\mathsf{nd}(A') \leq t_1$ and $TAct(A') \subseteq A_1$, and $bs' \ll bs$ then $bs' \cdot A' \ll bs \cdot (A_1 a_1 t_1 \cdot b')$.

Let us note that the symbol $\ll$ is overloaded, it is used to relate barbs, set of barbs, and processes. $\square$

Intuitively, a b-trace bs is *worse* than another one bs' if the actions that appear in both b-traces are the same, and the intermediate sets A_i that appear in bs are smaller than the ones appearing in bs'. For barbs, we must notice that whenever a process is in an undefined state, that is $t = \mathsf{nd}(A) < \infty$, it cannot pass any test *after* that time.

We extend the preorder $\ll$ to sets of barbs as follows:

$$B' \ll B \quad \text{iff} \quad \forall b \in B\, \exists b' \in B' \;:\; b' \ll b$$

The relation $\ll$ between sets of barbs induces a relation between processes, that is, $P \ll Q$ iff $\mathsf{Barb}(P) \ll \mathsf{Barb}(Q)$. The following example illustrates this.

Example 7.1. Let us consider the $RTPA_{core}$ processes

$$P = (\tau 0 \rightarrow \mathtt{STOP}) + (\tau 0 \rightarrow (a1 \rightarrow \mathtt{STOP} + b2 \rightarrow \mathtt{STOP}))$$
$$Q = (\tau 0 \rightarrow \mathtt{STOP}) + (\tau 0 \rightarrow (a1 \rightarrow \mathtt{STOP} + b2 \rightarrow \mathtt{STOP})) + (\tau 0 \rightarrow b2 \rightarrow \mathtt{STOP})$$

Their set of barbs are given by

$$\mathsf{Barb}(P) = \left\{ \begin{array}{l} \varnothing,\ \{a1, b2\}, \\ \varnothing a1 \varnothing,\ \{a1\} b2 \varnothing \end{array} \right\} \qquad \mathsf{Barb}(Q) = \left\{ \begin{array}{l} \varnothing,\ \{b2\}, \{a1, b2\}, \\ \varnothing a1 \varnothing,\ \varnothing b2 \varnothing,\ \{a1\} b2 \varnothing \end{array} \right\}$$

We have $Q \ll P$. But for $b = \varnothing b_2 \varnothing \in \mathsf{Barb}(Q)$ there is no barb $b' \in \mathsf{Barb}(P)$ such that $b \ll b'$. So, we have $P \not\ll Q$. In $\mathsf{Barb}(P)$, b at time 2 is *always offered* together with a at time 1. If P can synchronize with a test that offers a at time 1 then b at time 2 is never executed. On the contrary, in Q, b at time 2 may be executed. In fact, if we consider the test

$$T = (a1 \rightarrow \mathtt{OK}) + (\tau 2 \rightarrow \mathtt{OK}) + (b2 \rightarrow \mathtt{STOP})$$

it is easy to check that P must T and Q $\not{\text{m}}$ust$T$. So, we can conclude $P \not\sqsubseteq_{\text{must}} Q$. $\square$

The rest of the section is devoted to prove the desired characterization, that is,

$$P \sqsubseteq_{\text{must}} Q \quad \text{iff} \quad P \ll Q$$

First we need the following result.

Lemma 7.1. Let P be a process such that $P \Downarrow$ and $P \stackrel{it'}{\not\longrightarrow}$ for $t' < t$. Then we have

- $S(P) = (S(P) \upharpoonright t) \cup (S(\mathsf{upd}(P, t)) + t)$.
- $A \in \mathcal{A}(\mathsf{upd}(P, t))$ iff $(S(P) \upharpoonright t) \cup (A + t) \in \mathcal{A}(P)$.
- If $bs \neq \epsilon$ then $\mathsf{upd}(P, t) \stackrel{bs}{\Longrightarrow} P'$ iff $P \xrightarrow{(S(P) \upharpoonright t, t) \sqcup bs} P'$. $\square$

Proof:
By structural induction we have $\mathsf{upd}(P,t) \xrightarrow{et'} P'$ iff $P \xrightarrow{e(t'+t)} P'$, and then the result is immediate. □

Lemma 7.2. Let P be a process and T be a test. We have

- Let $bs = A_1 a_1 t_1 \cdots A_n a_n t_n$ and $bs' = A'_1 a_1 t_1 \cdots A'_n a_n t_n$ be b-traces such that $A_i \cap A'_i = \varnothing$. If $T \stackrel{bs}{\Longrightarrow} T'$ and $P \stackrel{bs'}{\Longrightarrow} P'$ then there exists a computation from $P \mid T$ to $P' \mid T'$.
- Let $A \in \mathcal{A}(P)$, and let us suppose that T has a computation $T = T_1 \xrightarrow{it_1} T_2 \xrightarrow{it_2} T_3 \cdots$ such that $A \cap ((S(T_i) \upharpoonright t_i) + t^i) = \varnothing$, where $t^i = \sum_{j=1}^{i-1} t_i$. Then,
  - If $t^i < \mathsf{nd}(A)$ then there exists a process P_i such that there is a computation from $P \mid T$ to $P_i \mid T_i$.
  - If $t^i < \mathsf{nd}(A)$ and $t^{i+1} \geq \mathsf{nd}(A)$, then there exists a process P' and a test T' such that $P' \Uparrow$, $T' = \mathsf{upd}(T_i, t')$ for some $t' \geq 0$, and there exists a computation from $P \mid T$ to $P' \mid T'$. □

Proof:
We have just to use the previous lemma and the following facts:

- If $P \xrightarrow{at} P'$, $T \xrightarrow{at} T'$ and $(S(P) \upharpoonright t) \cap (S(T) \upharpoonright t) = \varnothing$ then $P \mid T \xrightarrow{at} P' \mid T'$.
- If $P \xrightarrow{it} P'$, $T \stackrel{it'}{\not\longrightarrow}$ with $t' < t$ and $(S(P) \upharpoonright t) \cap S(T) = \varnothing$ then $P \mid T \xrightarrow{it} P' \mid \mathsf{upd}(T,t)$
- If $T \xrightarrow{it} T'$, $P \stackrel{it'}{\not\longrightarrow}$ with $t' < t$ and $(S(T) \upharpoonright t) \cap S(P) = \varnothing$ then $P \mid T \xrightarrow{it} \mathsf{upd}(P,t) \mid T'$

□

Now we can already prove the left to right side of the characterization:

Theorem 7.1. If $P \ll Q$ then $P \sqsubseteq_{\text{must}} Q$. □

Proof:
Let T be a test such that P must T. In order to check that Q must T, let us consider any complete computation of $Q \mid T$

$$Q \mid T = Q_0 \mid T_0 \xrightarrow{\tau t_1} Q_1 \mid T'_1 \cdots \xrightarrow{\tau t_k} Q'_k \mid T'_k \cdots$$

This computation may be unzipped into a computation of Q

$$Q = Q_{11} \xrightarrow{it^Q_{11}} \cdots Q_{1\,m_1-1} \xrightarrow{\tau t^Q_{1\,m_1-1}} Q_{1m_1} \xrightarrow{a_1 t^Q_{1m_1}} Q_{21} \cdots$$

and a computation of T

$$T = T_{11} \xrightarrow{it^T_{11}} \cdots T_{1\,n_1-1} \xrightarrow{\tau t^T_{1\,n_1-1}} T_{1n_1} \xrightarrow{a_1 t^T_{1n_1}} T_{21} \cdots$$

where $t_i = \sum_{j=1}^{m_i} t_{ij}^Q = \sum_{j=1}^{n_i} t_{ij}^T$. From the computations of Q and T we can get a sequence of b-traces of Q and T

$$bs_i^Q = A_1^Q a_1 t_1 \cdots A_i^Q a_i t_i, \qquad bs_i^T = A_1^T a_1 t_1 \cdots A_i^T a_i t_i,$$

such that $Q \overset{bs_i^Q}{\Longrightarrow} Q_{(i+1)1}$ and $T \overset{bs_i^T}{\Longrightarrow} T_{(i+1)1}$, where

$$A_j^Q = \bigcup_{k=1}^{m_j} \left((S(Q_{jk}) \upharpoonright t_{jk}^Q) + \sum_{l=1}^{k-1} t_{jl}^Q \right), \qquad A_j^T = \bigcup_{k=1}^{n_j} \left((S(T_{jk}) \upharpoonright t_{jk}^T) + \sum_{l=1}^{k-1} t_{jl}^T \right).$$

As the computation of P and T is possible, we have that $A_j^Q \cap A_j^T = \varnothing$. Given the b-trace bs_i^Q, there exists a state A such that $b_i = bs_i^Q \cdot A \in \mathsf{Barb}(Q)$. So, there exists $b = bs \cdot A \in \mathsf{Barb}(P)$ such that $b \ll b_i$. If $\mathsf{lon}(b) < \mathsf{lon}(b_i)$, by applying the previous lemma, we could find an unsuccessful computation of $P \mid T$, against our hypothesis. So, we have $\mathsf{lon}(b) = \mathsf{lon}(b_i)$ and then there exists a process P_{i+1} such that there exists a computation from $P \mid T$ to $P_{i+1} \mid T_{(i+1)1}$.

Let us suppose that the sequence of b-traces is infinite. Then for any $k \in \mathbb{N}$ there exists a process P_k such that there exists a computation from $P \mid T$ to $P_k \mid T_{k\,1}$. As the operational semantics is finite branching, by applying Koning's lemma, we have that there exists an infinite computation in which all the T_{ij}'s appears. Since P must T we have that there exists T_{ij} such that $T_{ij} \xrightarrow{\texttt{OK}}$. So, the computation of $Q \mid T$ is successful.

Now, let us suppose that there exists a *last* b-trace bs_k^Q. Then, there exists a state A^Q such that

$$A^Q \upharpoonright t^{Q\,j} = \bigcup_{i=1}^{j-1} \left((S(Q_{(k+1)i} \upharpoonright t_{(k+1)\,i}^Q) + \sum_{l=1}^{i-1} t^{Q\,i} \right) \quad \text{where } t^{Q\,i} = \sum_{l=1}^{i-1} t_{(k+1)l}^Q$$

and $b^Q = bs_k^Q \cdot A^Q \in \mathsf{Barb}(Q)$. As $P \ll Q$, there exists some $b^P = bs^P \cdot A^P \in \mathsf{Barb}(P)$ such that $b^P \ll b^Q$. Let us suppose that

$$P \overset{bs^P}{\Longrightarrow} P' \quad \text{and} \quad A^P \in \mathcal{A}(P')$$

If $\mathsf{lon}(b^P) < \mathsf{lon}(b^Q)$, by applying the previous lemma we can find an unsuccessful computation of $P|T$, against our hypothesis. So we must have $\mathsf{lon}(b^P) = \mathsf{lon}(b^Q)$. Let us consider

$$t^{T,i} = \sum_{j=1}^{i-1} t_{(k+1)j}^T$$

then, for any $t^{T,i} < \mathsf{nd}(A^P)$, there exists a computation from $P' \mid T_{(k+1)1}$ to $P_i \mid T_{(k+1)i}$ for some P_i. If either $\mathsf{nd}(A^P) < \infty$ or the collection of tests is finite, as P must T there must be some test T_{ij} such that $T_{ij} \xrightarrow{\texttt{OK}}$; so we have that the computation of $Q \mid T$ is successful. If $\mathsf{nd}(A^P) = \infty$ and the collection of tests is infinite, as the operational semantics is finite branching, by applying Kï¿½ning's lemma, we have that there must be some infinite computation, in which all the T_{ij}'s appear. As P must T we have that there exists T_{ij} such that $T_{ij} \xrightarrow{\texttt{OK}}$, so the computation of $Q \mid T$ is successful. □

Now we have to prove that $P \sqsubseteq_{\text{must}} Q$ implies $P \ll Q$. First, let us assume that $P \not\ll Q$. Then, by definition, there exists some $b \in \mathsf{Barb}(Q)$ such that there does not exist any $b' \in \mathsf{Barb}(P)$ such that $b' \ll b$. In order to find a test T such that P must T and Q $\not\text{must}\, T$, we generalize the procedure described in example 7.1. First, we need some auxiliary definitions.

Definition 7.2. Let B be a set of barbs and bs a b-trace. We define the barbs of B *after* bs as:

$$\mathsf{Barb}(B, bs) = \{b \mid bs \cdot b \in B\}$$

Given a barb b and a set of barbs B such that there does not exist a barb $b' \in B$ with $b' \ll b$, we say that a test T is *well formed with respect to* B *and* b when it can be derived by applying the following rules:

- If $b = A$ we take a finite set $A_1 \subseteq TAct$ such that for all $A' \in B$ with $\mathsf{nd}(A') \leq \mathsf{nd}(A)$, we have $A_1 \cap A' \neq \varnothing$. Then, by considering

$$T_1 = \begin{cases} it \rightarrow \mathtt{OK} & \text{if } \mathsf{nd}(A) = t < \infty \\ \mathtt{STOP} & \text{otherwise} \end{cases} \quad \text{and} \quad T_2 = \begin{cases} \displaystyle\sum_{at \in A_1} at \rightarrow \mathtt{OK} & \text{if } A_1 \neq \varnothing \\ \\ \mathtt{STOP} & \text{otherwise} \end{cases}$$

  we have that $T = T_1 + T_2$ is *well formed with respect to* B *and* b.

- If $b = Aat \cdot b_1$ we consider any finite set $A_1 \subseteq TAct$ such that $A_1 \cap A = \varnothing$ and any barb $A'at \cdot b_1' \in B$ satisfying either $A' \cap A \neq \varnothing$ or $b_1' \not\ll b_1$. Then, we consider the test

$$T_2 = \begin{cases} \displaystyle\sum_{at \in A_1 \setminus A} at \rightarrow \mathtt{OK} & \text{if } A_1 \setminus A \neq \varnothing \\ \\ \mathtt{STOP} & \text{otherwise} \end{cases}$$

  and the set of barbs $B_1 = \{b' \mid A' \subseteq A \text{ and } A'at \cdot b' \in B\}$. If $B_1 \neq \varnothing$ then we can take as T_1 any well formed test with respect to B_1 and b_1; otherwise let $T_1 = \mathtt{STOP}$. Then $T = T_1 + T_2 + it \rightarrow \mathtt{OK}$ is a *well formed test with respect to* B *and* b. □

It is possible that for a given set of barbs B and a barb b, there does not exist a well formed test T with respect to B and b, because it might not exist the finite set A_1 required in the first part of the definition. But, as the operational semantics is finite branching, for any $B = \mathsf{Barb}(P)$ and $b \in \mathsf{Barb}(Q)$ such that there is no $b' \in B$ with $b' \ll b$, there exists a well formed test T with respect to B and b.

Proposition 7.1. Let T be a well formed test with respect to a set of barbs B and a barb b. Then,

- If $b \in \mathsf{Barb}(Q)$ then Q $\not\text{must}\, T$.
- If $B = \mathsf{Barb}(P)$ and there is no $b' \in B$ such that $b' \ll b$, then P must T. □

As a corollary of the previous result we have that the right to left side of the characterization holds

Theorem 7.2. Let P and Q be processes. If $P \sqsubseteq_{\rm must} Q$ then $P \ll Q$. □

Proof:
Let us suppose that $P \not\ll Q$. Then, there exists a barb $b \in \mathsf{Barb}(Q)$ such that there does not exists $b' \in \mathsf{Barb}(P)$ verifying that $b' \ll b$. So, we can find a well formed test T with respect to $\mathsf{Barb}(P)$ and b. Then, by the previous proposition we have that P must T and Q $\not\text{must}\, T$. So, $P \not\sqsubseteq_{\rm must} Q$, that contradicts our hypothesis. □

8. May testing semantics

In an untimed framework *may* semantics is simply equivalent to just trace semantics [14]. But when time is introduced the *may* semantics also becomes more complex, since, as usual, we are assuming that internal actions are executed as soon they are available. For that reason it is possible to detect by means of tests not only the actions that have been executed, but also those that could have been chosen instead. To be exact, we will prove that for non-divergent processes the *may* testing semantics is equivalent to the *must* testing semantics. This is so because for any adequate test we can define a dual one in such a way that a process passes the original test in the *must* sense if and only if it does not pass the dual one in the *may* sense.

The situation is much more complicated in the case of non-divergent processes. It is well known that in the untimed case by using *may* testing we can (partially) know the possible behaviors of a process after the instant at which it diverges, what is not possible under *must* semantics. This is also the case in the timed case. In fact, we will present a couple of examples showing that in the general case *may* and *must* testing orderings are incomparable. Hence, another characterization *must* be given for the *may* testing semantics in the general case including divergent processes.

8.1. Divergence free processes

First, we will study the relationship between *may* and *must* testing semantics in the case when the involved processes are divergence free. We will show that in this case, rather surprisingly, both relations are symmetric, that is, we have

$$P \sqsubseteq_{\rm must} Q \quad \text{iff} \quad Q \sqsubseteq_{\rm may} P$$

Even more, we will show that *may* testing can be viewed as the *dual* relation of *must* testing in the sense that for any *standard test* T characterizing *must* semantics we can define its *dual* T^* in a very simple and natural way. We will show that a process passes a family of tests in the *must* sense iff it does not pass the corresponding *dual tests* in the *may* sense.

Let us recall that a process P is *divergent*, $P \Uparrow$, if P can execute in a row infinitely many internal actions, all of them at (local) time 0. We say that P is *divergence free* when no execution of P leads to a divergent process. For instance, the process $P = a1 \rightarrow \texttt{STOP}$ is divergence free, while $Q = a1 \rightarrow \texttt{DIV}$ is not, because of the existence of the computation $Q \xrightarrow{a1} \texttt{DIV}$. We could formally define divergence freedom directly from the operational semantics, but since this definition is a little cumbersome, we present instead the equivalent characterization in terms of barbs.

Definition 8.1. We say that a process P is *divergence free* iff for each barb $bs \cdot A \in \mathsf{Barb}(P)$ we have $\mathsf{nd}(A) = \infty$. □

First, we will prove the left to right implication: $P \sqsubseteq_{\text{must}} Q$ implies $Q \sqsubseteq_{\text{may}} P$. We will use the following characterization of the *must* testing order from the previous section: $P \sqsubseteq_{\text{must}} Q$ iff $\mathsf{Barb}(P) \ll \mathsf{Barb}(Q)$. So it is enough the following result.

Proposition 8.1. Let P and Q be processes. We have $\mathsf{Barb}(Q) \ll \mathsf{Barb}(P)$ implies $P \sqsubseteq_{\text{may}} Q$ □

Proof:
Let T be a test such that P may T. Then, there exist some process P' and some test T' such that $T' \xrightarrow{\texttt{OK}}$ and there exists a computation from $P \mid T$ to $P' \mid T'$. So, there exist P'' and T'' and a couple of b-traces $bs = A_1 a_1 t_1 \cdots A_n a_n t_n$ and $bs' = A'_1 a_1 t_1 \cdots A'_n a_n t_n$ such that $P \overset{bs}{\Longrightarrow} P''$, $T \overset{bs'}{\Longrightarrow} T''$, $A_i \cap A'_i = \varnothing$, and there exists a computation from $P \mid T$ to $P'' \mid T''$ and another one from $P'' \mid T''$ to $P' \mid T'$.

Now, from the computations of P'' and T'' to P' and T' we get two states $A \in \mathcal{A}(P'')$ and $A' \in \mathcal{A}(T'')$ satisfying $A \cap A' = \varnothing$.

Since $\mathsf{Barb}(Q) \ll \mathsf{Barb}(P)$ and P and Q are divergence free, there exist some barb $b'' = bs'' \cdot A'' \in \mathsf{Barb}(Q)$ with $bs'' = A''_1 a_1 t_1 \cdots A''_n a_n t_n$ and some processes Q' and Q'' such that $A''_i \subseteq A_i$, $A'' \subseteq A$, $Q \overset{bs''}{\Longrightarrow} Q''$ and there is a computation from Q'' to Q', that generates the state A'' by applying Proposition 6.1. So we have $A''_i \cap A'_i = \varnothing$ and $A'' \cap A' = \varnothing$, so we have a computation from $Q \mid T$ to $Q'' \mid T''$. Since Q is divergence free, we also obtain a computation from $Q'' \mid T''$ to $Q' \mid T'$. □

Next we prove the right to left side implication. For it, we will adapt the family of *standard tests* characterizing *must* semantics from the previous section. That is, in order to prove $P \not\sqsubseteq_{\text{must}} Q$ we can find a test T such that P must T but Q $\not$must T. These tests are similar, but not exactly the same, to those presented in the previous section. Although in order to relate $\sqsubseteq_{\text{may}}$ and $\sqsubseteq_{\text{must}}$ we could also use here those tests, we have preferred to use instead this new definition, because by considering the corresponding *dual tests* we can directly obtain that relationship, thus emphasizing the duality between *must* and *may* passing of tests. The tests that constitute the new family of *standard tests* are those obtained by applying the following definition.

Definition 8.2. Given a barb b and a set of barbs B with there is no barb $b' \in B$ such that $b' \ll b$, we say that a test T is *may well formed with respect to B and b* when it can be derived by applying the following rules:

- If $b = A$ then we take any finite set $A_1 \subseteq Act \times \mathcal{T}$ such that for all $A' \in B$ we have $A_1 \cap A' \neq \varnothing$. Then, by considering

$$T_2 = \Big(\sum_{at \in A_1} at \to \texttt{OK}\Big) + \tau t \to \texttt{STOP}\,, \quad \text{where } t > \max\{t' \mid at' \in A_1\}$$

we have that $T = T_1 + T_2$ is *may well formed with respect to B and b*.

- If $b = Aat \cdot b_1$ then we consider any finite set $A_1 \subseteq Act \times \mathcal{T}$ with $A_1 \cap A = \varnothing$ and such that all barb $A'at \cdot b'_1 \in B$ either satisfies $A' \cap A_1 \neq \varnothing$ or $b'_1 \not\ll b_1$. When $A_1 \neq \varnothing$, we consider the test

$$T_2 = \sum_{at \in A_1 \setminus A} at \to \texttt{OK}.$$

Besides, by taking the set of barbs $B_1 = \{b' \mid A' \subseteq A \text{ and } A'at \cdot b' \in B\}$, when $B_1 \neq \varnothing$, we can take as T_1 any well formed test with respect to B_1 and b_1. Then we have that

$$T = \begin{cases} T_1 + T_2 + it \to \mathtt{OK} & \text{if } A_1 \neq \varnothing,\ B_1 \neq \varnothing \\ T_1 & \text{if } A_1 \neq \varnothing,\ B_1 = \varnothing \\ T_2 + it \to \mathtt{OK} & \text{if } A_1 = \varnothing,\ B_1 \neq \varnothing \end{cases}$$

is a *may well formed test with respect to B and b*.

□

Given an arbitrary set of barbs B and a barb b, it is possible that there does not exist may well formed test T with respect to B and b because the finite set A_1 required in the first part of the definition might not exist. But, as the operational semantics is finitely branching, for all $B = \mathsf{Barb}(P)$ and $b \in \mathsf{Barb}(Q)$ such that there is no $b' \in B$ with $b' \ll b$, there is some well formed test T with respect to B and b. Finally, dual tests are defined as expected.

Definition 8.3. Let T be a test. We define its dual test, denoted by T^*, by interchanging the tailing occurrences of `STOP` and `OK` in T. □

The well formed tests and their duals satisfy the following:

Proposition 8.2. Let B be a set of barbs and b be a barb such that and there is no $b' \in B$ with $b' \ll b$. Let us consider a *may well formed test with respect to B and b T*. Then,

$$\begin{array}{rcl} b \in \mathsf{Barb}(Q) & \text{implies} & Q \not\text{must}\, T \text{ and } Q \text{ may } T^*. \\ B = \mathsf{Barb}(P) & \text{implies} & P \text{ must } T \text{ and } P \not\text{may}\, T^*. \end{array}$$

□

Proposition 8.3. Let P and Q be processes. We have $P \sqsubseteq_{\text{may}} Q$ implies $Q \sqsubseteq_{\text{must}} P$. □

Proof:
Let us suppose that $Q \not\sqsubseteq_{\text{must}} P$. Then, there must exist some $b \in \mathsf{Barb}(P)$ such that there is no $b' \in \mathsf{Barb}(Q)$ verifying $b' \ll b$. Then we take a well formed test T with respect to $\mathsf{Barb}(Q)$ and b such that Q must T and P $\not\text{must}\, T$. Then, by applying the previous proposition, we have P may T^* and Q $\not\text{may}\, T^*$. This contradicts our hypothesis, $P \sqsubseteq_{\text{may}} Q$. □

Finally, as a consequence of Propositions 8.1 and 8.3, we obtain the *may testing characterization theorem* for divergence-free processes that we were looking for:

Theorem 8.1. Let P and Q be processes. We have $P \sqsubseteq_{\text{may}} Q$ iff $Q \sqsubseteq_{\text{must}} P$ □

8.2. Processes with divergences

Once we have studied the relation $\sqsubseteq_{\text{may}}$ for non divergent processes we proceed to study it in the general case. Since in the untimed case we already had $\simeq_{\text{must}} \neq \simeq_{\text{may}}$ when divergences appear, we could expect that in the timed setting we would have the same result. This is the case, as the following example shows.

Example 8.1. Let us consider the processes $P = \mathtt{DIV}$ and $Q = \tau 0 \rightarrow \mathtt{DIV} + \tau 0 \rightarrow a1 \rightarrow \mathtt{STOP}$. It is straight forward to show that P and Q are equivalent under *must* testing, that is, $P \sqsubseteq_{\text{must}} Q$ and $Q \sqsubseteq_{\text{must}} P$. On the other hand, we have $Q \not\sqsubseteq_{\text{may}} P$.

We also have $\simeq_{\text{must}} \not\supseteq \simeq_{\text{may}}$, what means that we really need divergence freedom in order to prove Proposition 8.3. To show it, let us consider the following processes:

$$P = \tau 0 \rightarrow ((a0 \rightarrow \mathtt{STOP}) + (\tau 2 \rightarrow \mathtt{DIV})) + \tau 0 \rightarrow ((a0 \rightarrow \mathtt{STOP}) + (b0 \rightarrow \mathtt{STOP}) + (\tau 1 \rightarrow \mathtt{DIV}))$$
$$Q = \tau 0 \rightarrow ((a0 \rightarrow \mathtt{STOP}) + (\tau 2 \rightarrow \mathtt{DIV})) + \tau 0 \rightarrow ((a0 \rightarrow \mathtt{STOP}) + (b0 \rightarrow \mathtt{STOP}) + (\tau 2 \rightarrow \mathtt{DIV}))$$

It is not difficult to show, by using Theorem 8.2, that under *may* testing semantics both processes are equivalent, that is, $P \simeq_{\text{may}} Q$. However, we have $Q \not\sqsubseteq_{\text{must}} P$, and so $P \not\simeq_{\text{must}} Q$. □

In order to characterize $\sqsubseteq_{\text{may}}$ in the general case, we introduce the following notation.

Definition 8.4. Let $bs \cdot A$ and $bs' \cdot A'$ be barbs. We write $bs' \cdot A' \ll_{\text{may}} bs \cdot A$ if $bs' \ll bs$, $\mathsf{nd}(A') \geq \mathsf{nd}(A)$, and $A' \upharpoonright \mathsf{nd}(A) \subseteq A$. We define the relation $\ll_{\text{may}}$ between sets of barbs, by writing $B_1 \ll_{\text{may}} B_2$ iff for all $b_1 \in B_1$ there exists $b_2 \in B_2$ such that $b_2 \ll_{\text{may}} b_1$. □

The rest of the section is devoted to prove the *May Characterization Theorem* for the general case: $P \sqsubseteq_{\text{may}} Q$ iff $\mathsf{Barb}(P) \ll_{\text{may}} \mathsf{Barb}(Q)$. It is an immediate consequence of Propositions 8.4 and 8.5 below. In the following, $\mathsf{t_{bs}}(bs)$ stands for the *duration* of bs, defined by taking $\mathsf{t_{bs}}(\epsilon) = 0$ and $\mathsf{t_{bs}}(A_1 a_1 t_1 \cdot bs_1) = t_1 + \mathsf{t_{bs}}(bs_1)$. To prove the left to right implication of the theorem, we need a special kind of tests.

Definition 8.5. Let b be a barb. We inductively define the test $T(b)$ as follows:

$$\begin{array}{rcll} T(\epsilon \cdot A) & = & \tau t \rightarrow \mathtt{OK} + \displaystyle\sum_{a't' \notin A,\ t' < t} a't' \rightarrow \mathtt{STOP} & \big(t = \mathsf{nd}(A)\big) \\ T(A'a't' \cdot b, t) & = & a't' \rightarrow T(b, a, t) + \displaystyle\sum_{a''t'' \notin A',\ t'' < t} a''t'' \rightarrow \mathtt{STOP} & \end{array}$$

□

For these distinguished tests it is easy to check the following property.

Lemma 8.1. Let P and Q be processes. P may $T(b)$ iff there exists $b' \in \mathsf{Barb}(P)$ such that $b' \ll_{\text{may}} b$. □

Proposition 8.4. Let P and Q be processes. $P \sqsubseteq_{\text{may}} Q$ implies $\mathsf{Barb}(P) \ll_{\text{may}} \mathsf{Barb}(Q)$. □

Proof:
Let us consider $b = bs \cdot A \in \mathsf{Barb}(P)$. Then, we have P may $T(b)$. Since $P \sqsubseteq_{\text{may}} Q$ we also have Q may $T(b)$, and by applying the previous lemma we get the desired result. □

Proposition 8.5. Let P and Q be processes. $\mathsf{Barb}(P) \ll_{\mathrm{may}} \mathsf{Barb}(Q)$ implies $P \sqsubseteq\!\!\!\sim_{\mathrm{may}} Q$. □

Proof:
The proof of this proposition is very similar to that of Proposition 8.1. The only difference is the way that the adequate barb $b'' = bs'' \cdot A'' \in \mathsf{Barb}(Q)$ is found. To do it in this case we have to take into account that $P \ll_{\mathrm{may}} Q$. From this fact, we can also conclude that this barb verifies $bs'' \ll bs$, $A'' \upharpoonright \mathsf{nd}(A) \subseteq A$, and $\mathsf{nd}(A'') \geq \mathsf{nd}(A)$. Then, for the corresponding processes Q' and Q'' we obtain a successful computation from $Q \mid T$ to $Q' \mid T'$. □

Theorem 8.2. Let P and Q be processes. $P \sqsubseteq\!\!\!\sim_{\mathrm{may}} Q$ iff $\mathsf{Barb}(P) \ll_{\mathrm{may}} \mathsf{Barb}(Q)$. □

9. Conclusions and Future Work

In this paper we have presented a suitable testing semantics for the process algebra $RTPA$. The syntax of this algebra is very rich and expressive. On the one hand, that expressiveness makes this algebra the ideal framework for a real time system programmer. On the other hand, it makes the algebra quite difficult to study from a formal point of view. We have overcome this problem by defining a core language that is simpler. However, this simpler language is powerful enough to represent to $RTPA$ process relations.

Testing semantics for process algebras are very intuitive: two process are equivalent if they present the same behavior when they interact with the environment. The definition of these semantics usually is rather direct. The real problem resides in applying that definition: it requires to take an infinite amount of tests to compare two processes. So, it is usually necessary to provide an alternative characterization to the original definition. That is what we have done in this paper for both the *must* testing semantics an the *may* testing semantics.

Another interesting result is the relationship between the *must* testing semantics and the *may* testing semantics. It turns out that, for a broad class of processes, both relations are *dual* one each other. This class of processes is the *divergence free processes*, that is, the class of processes that do not present abnormal behavior.

As future work, following [14], the characterization of the testing semantics could be used to provide a denotational semantics to $RTPA$. That denotational semantics would be correct and complete with respect the testing semantics presented here. It could be interesting to compare the denotational semantics obtained in this way with the one that has already been defined for $RTPA$ in [35].

References

[1] J.C.M. Baeten and C.A. Middelburg. *Process algebra with timing*. EATCS Monograph. Springer, 2002.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Computer Science 18. Cambridge University Press, 1990.

[3] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North Holland, 2001.

[4] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202(1-2):1–54, 1998.

[5] M. Bravetti and R. Gorrieri. The theory of interactive generalized semi-Markov processes. *Theoretical Computer Science*, 282(1):5–32, 2002.

[6] D. Cazorla, F. Cuartero, V. Valero, F.L. Pelayo, and J.J. Pardo. Algebraic theory of probabilistic and non-deterministic processes. *Journal of Logic and Algebraic Programming*, 55(1–2):57–103, 2003.

[7] R. Cleaveland, Z. Dayar, S.A. Smolka, and S. Yuen. Testing preorders for probabilistic processes. *Information and Computation*, 154(2):93–148, 1999.

[8] J. Davies and S. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.

[9] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[10] R. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1995.

[11] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In *16th Int. Symp. on Computer Performance Modelling, Measurement and Evaluation, PERFORMANCE'93, LNCS 729*, pages 121–146. Springer, 1993.

[12] J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[13] P.G. Harrison and B. Strulo. SPADES – a process algebra for discrete event simulation. *Journal of Logic Computation*, 10(1):3–42, 2000.

[14] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[15] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[16] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[17] L. Llana and D. de Frutos. Denotational semantics for timed testing. In *4th AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software, LNCS 1231*, pages 368–382, 1997.

[18] L. Llana and D. de Frutos. Relating may and must testing semantics for discrete timed process algebras. In *5th Asian Computing Science Conference, ASIAN'99, LNCS 1742*, pages 74–86. Springer, 1999.

[19] N. López and M. Núñez. A testing theory for generally distributed stochastic processes. In *12th Int. Conf. on Concurrency Theory, CONCUR'01, LNCS 2154*, pages 321–335. Springer, 2001.

[20] N. López, M. Núñez, and F.L. Pelayo. STOPA: A STOchastic Process Algebra for the formal representation of cognitive systems. In *3rd IEEE Int. Conf. on Cognitive Informatics, ICCI'04*, pages 64–73. IEEE Computer Society Press, 2004.

[21] N. López, M. Núñez, and F.L. Pelayo. Specifying the memorization process with STOPA. *The International Journal of Cognitive Informatics & Natural Intelligence*, 1(4):47–60, 2007.

[22] N. López, M. Núñez, I. Rodríguez, and F. Rubio. A formal framework for e-barter based on microeconomic theory and process algebras. In *Innovative Internet Computer Systems, LNCS 2346*, pages 217–228. Springer, 2002.

[23] N. López, M. Núñez, and F. Rubio. An integrated framework for the analysis of asynchronous communicating stochastic processes. *Formal Aspects of Computing*, 16(3):238–262, 2004.

[24] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[25] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[26] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *3rd Int. Conf. on Computer Aided Verification, CAV'91, LNCS 575*, pages 376–398. Springer, 1991.

[27] M. Núñez. Algebraic theory of probabilistic processes. *Journal of Logic and Algebraic Programming*, 56(1–2):117–177, 2003.

[28] M. Núñez and D. de Frutos. Testing semantics for probabilistic LOTOS. In *8th IFIP WG6.1 Int. Conf. on Formal Description Techniques, FORTE'95*, pages 365–380. Chapman & Hall, 1995.

[29] M. Núñez, D. de Frutos, and L. Llana. Acceptance trees for probabilistic processes. In *6th Int. Conf. on Concurrency Theory, CONCUR'95, LNCS 962*, pages 249–263. Springer, 1995.

[30] M. Núñez and I. Rodríguez. PAMR: A process algebra for the management of resources in concurrent systems. In *21st IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'01*, pages 169–185. Kluwer Academic Publishers, 2001.

[31] M. Núñez, I. Rodríguez, and F. Rubio. Formal specification of multi-agent e-barter systems. *Science of Computer Programming*, 57(2):187–216, 2005.

[32] F. L. Pelayo, F. Cuartero, V. Valero, and D. Cazorla. An example of performance evaluation by using the stochastic process algebra ROSA. In *7th Int. Conf. on Real-Time Systems and Applications*, pages 271–278. IEEE Computer Society Press, 2000.

[33] F.L. Pelayo, M. Núñez, and N. López. Specifying the memorization process with STOPA. In *4th IEEE Int. Conf. on Cognitive Informatics, ICCI'05*, pages 238–247. IEEE Computer Society Press, 2005.

[34] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[35] Xinming Tan and Yingxu Wang. A denotational semantics for RTPA. *Electrical and Computer Engineering, 2005. Canadian Conference on*, pages 2057–2060, May 2005.

[36] Y. Wang. On cognitive informatics. In *1st IEEE Int. Conf. on Cognitive Informatics, ICCI'02*, pages 34–42. IEEE Computer Society Press, 2002.

[37] Y. Wang. The Real Time Process Algebra (RTPA). *Annals of Software Engineering*, 14:235–274, 2002.

[38] Y. Wang. Cognitive informatics: A new transdisciplinary research field. *Brain and Mind*, 4:115–127, 2003.

[39] Y. Wang. Using process algebra to describe human and software behaviors. *Brain and Mind*, 4:199–213, 2003.

[40] Y. Wang. On the mathematical laws of software. In *18th Canadian Conf. on Electrical and Computer Engineering, CCECE'05*, pages 1086–1089, 2005.

[41] Y. Wang. Cognitive informatics and contemporary mathematics for knowledge manipulation. In Guoyin Wang, James F. Peters, Andrzej Skowron, and Yiyu Yao, editors, *RSKT*, volume 4062 of *Lecture Notes in Computer Science*, pages 69–78. Springer, 2006.

[42] Y. Wang. The theoretical framework of cognitive informatics. *The International Journal of Cognitive Informatics & Natural Intelligence*, 1(1):1–27, 2007.

[43] Y. Wang, L. Dong, and G. Ruhe. Formal description of the cognitive process of decision making. In *3rd IEEE Int. Conf. on Cognitive Informatics, ICCI'04*, pages 124–130. IEEE Computer Society Press, 2004.

[44] Y. Wang and Y. Wang. Recent advances in cognitive informatics. *IEEE Transactions on Systems, Man, and Cybernetics C*, 36(2):121–123, 2006.

[45] W. Yi. CCS+ Time = an interleaving model for real time systems. In *18th Int. Colloquium on Automata, Languages and Programming, ICALP'91, LNCS 510*, pages 217–228. Springer, 1991.