From Timethreads to LOTOS: A First Pass

Daniel Amyot

Report SCE-93-38
December 8, 1993

Department of Systems and Computer Engineering
Carleton University
Ottawa Canada K1S 5B6
email: damyot@csi.uottawa.ca
tel: (613) 564-9439
fax: (613) 564-9486

This work has been supervised by Professor R.J.A. Buhr (U. Carleton) and Professor L. Logrippo (Ottawa U.).

From Timethreads to LOTOS: A First Pass

Daniel Amyot
Telecommunication Software Engineering Research Group
University of Ottawa
Department of Computer Science
Ottawa, Ont. Canada K1S 9B4
email: damyot@csi.uottawa.ca

Abstract

A LOTOS interpretation of the Timethread notation is given in this document. Timethreads are an informal technique that focuses on end-to-end behaviours of a system while LOTOS is a formal technique with well developed theories of transformation, validation and testing. Timethreads constructors and their interpretation are discussed in the context of their possible use as a thinking tool in a design framework.

Keywords: Design methodology, Formal Description Techniques, LOTOS, Timethreads

1 Introduction

1.1 Motivation

Formal methods provide many ways to transform, validate, verify, and test a specification. However, they are limited in their use mostly because they are often not justified on a cost benefit analysis and they are seen as being rather mathematical and intellectually hard to use [Tur 93].

Engineers using formal methods may find too long the step from requirements to specification. They can lose trace of some basic end-to-end behaviours, while being in their design process. Visualization is also essential from a design perspective, and most formal methods do not provide such facility.

Where formal methods alone fail, a combination of engineering principles and formal methods may lead to rigorous and cost-effective computer system design.

1.2 The Timethread Notation

"A timethread is a path traced over a system to show operational behaviour. They are useful for design discovery and system reasoning at a global perspective" [LaB 92]. A timethread links activities performed by the system, resulting from some stimulus (cause) and terminating with some eventual response (effect). We use timethreads to stay focused on the end-to-end behaviour of the system we want to design.

This notation is considered intuitive and appealing by many engineers. At a very abstract level, timethreads leave intentionally many details unresolved. Refinement permits to clarify many of these details.

Still very young and not really formal, the Timethread notation has many open-ended problems. This document tries to provide a better semantics to this notation and to solve some of its problems, while keeping it appealing to engineers.

1.3 Why LOTOS?

LOTOS [ISO88] is a standardized Formal Description Technique that has been chosen as a formal basis for our Timethread notation. Other formal methods, e.g., Petri nets [FCB 93] and event structures, could be considered as options, but some problems are associated with these techniques:

Petri nets: Petri nets are a well-known specification technique. One of the major drawbacks of Petri nets, particularly in a system design perspective, is the complexity of compositionality features that are available. Nowadays, to solve these modularity and compositionality problems, enhancements to the basic model are provided, such as using a state machine as the basic component in the theory of modular Petri nets. Nevertheless, a few problems still exist: modularity is achieved only for limited classes of nets and behavioral properties are still very hard to analyse. Correctness preserving transformations are being developed, but much work still remains to be done in this area. These points are developed in [BDC 92].

Event structures: Event structure are a poset based model for describing the behaviour of distributed systems. They have an independent theory rooted in the theory of formal languages, but are too weak to express more general nets and problems, such as the producer and consumer paradigm (for example, see [Roz 92]). Extensions have been proposed and could seem as powerful as labelled transition systems (LTS), which are LOTOS' underlying model. However, there is no standardized event structures-based language available for the moment, and LTS, which are well understood, can be considered sufficient for our work.

What is very interesting in LOTOS is that we can manipulate, combine, factor, and transform various expressions quite easily. Many properties can be verified, tested and validated by the numerous available tools [GLO 91]. Extensions, although not standardized yet, will eventually provide other functionalities and facilities very useful for real-time and distributed systems design.

Previous work has been done on similar approaches. In [ViB 91], the authors presented a technique that can be used to support an effective process for generating the design of concurrent systems, with the help of timethreads (called *slices* in the paper) and LOTOS. In [LaB 92], the authors try to see if two different approaches of a design conception, ObjecTime[1] and LOTOS, could be used in a complementary way in order to add timethreads concepts to ObjecTime. The approach presented in the current document differs considerably from these two but the experience gained from [ViB 91] and [LaB 92] will help in avoiding some mistakes.

1.4 Organization of the Document

Section 2 introduces some definitions and a *sequence of activities* notation used in most examples. Section 3 presents many concepts related to LOTOS interpretation of timethreads: a basic timethread set, levels of specifications, and instance identifiers. Section 4 defines the basic LOTOS interpretation of single timethreads, while section 5 deals with simple timethread interactions. Section 6 mentions some special symbols that are part of in the Timethread notation. In section 7, we discuss a number of topics related to a more complete design methodology. Finally, a conclusion is given in section 8.

2 Definitions and Notation

2.1 Definitions

Since LOTOS and the Timethread notation use some common words with different meanings, we define a specific terminology that will be used in this document:

Timethread: Cause-to-effect relationship.

Triggering event: Starting event of a timethread.

Resulting event: Ending event, termination of a timethread.

1. ObjecTime is the new name for the real-time CASE toolset Telos.

Process: By default, will be used to denote a LOTOS behaviour abstraction, except where the meaning is expressed explicitly as in *Machine Charts process* and *design process*.

Interaction: General relation of observation between the environment and a triggering or resulting event, or between many timethreads on a waiting place.

Synchronization: Special case of interaction, usually artificial and internal, within one timethread. Multiway synchronization refers however to the LOTOS concept.

Activity: Action or event along a timethread.

Event: Activity on which there is interaction. Events are of three kinds: triggering, resulting or synchronization events.

Action: Activity on which no timethread interaction is allowed. An action corresponds to a certain functionality within the system.

2.2 Notation Used

In the following examples, a special notation to represent sequences of activities and LOTOS behaviours will be used:

- L is the alphabet, or the set of activities (including the internal action **i**).
- A^+ is a non-empty sequence of activities with the BNF:
  $A^+ ::= a \mid a; A^+$.
- A is a sequence of activities with the following BNF notation:
  $A ::= \varnothing \mid A^+$, where $\varnothing$ represents the empty sequence.
- B is a LOTOS behaviour expression. For instance, a sequence A followed by ***stop***, ***exit*** or a process instantiation is a behaviour expression.
- P, Q and R are usually used as LOTOS process identifiers.

Most LOTOS process definitions in our examples will not include gate parameters for conciseness. Note also that, in a behaviour expression:

- $\varnothing$; ***exit*** reduces to ***exit***
- $\varnothing$; ***stop*** reduces to ***stop***
- $\varnothing$; P reduces to P (where P is a process instantiation)

These equivalences are given to match LOTOS' syntax.

New words or concepts, as well as references to timethreads and LOTOS code, will be *italicized.*

3 Basic Concepts of Timethreads in LOTOS

3.1 Basic Timethread Set and Notation Elements

The Timethread notation [BuC 93][Buh 93a] includes the basic set of timethreads, possible interactions between timethreads and special symbols. Sections 4, 5 and 6 of the current document will present the LOTOS semantics of these timethreads and we will see that, from a LOTOS point of view, many details will become more abstract, reducing the complexity of the interpretation. Basically, there will be one LOTOS process per timethread. However, extra processes may be created to accommodate asynchronous or concurrent behaviours.

Figure 1 shows the basic notation elements of timethreads:

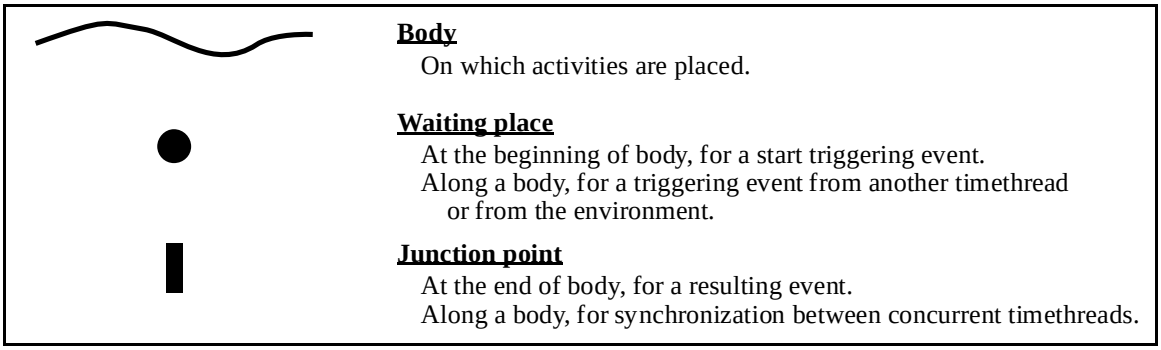

Figure 1 : Basic notation elements

Waiting places and junction points will be represented as LOTOS gates on which interaction with the environment or with other timethreads will occur. The body will only represent the sequencing of activities along it.

Note that some LOTOS operators, like the disabling and the enabling, will be put aside at this stage for simplicity reasons. Abstract data types and their corresponding operators will also be put aside. We try to use only a small set of LOTOS operators to be able to answer questions asked to a specification obtained from timethreads.

For instance, timethread interactions, which could be considered a case of enabling, can be easily generalized in LOTOS with multiway synchronization on extra hidden gates. We try not to use the LOTOS enabling operator (>>) since most of its functionality can be simulated with synchronization on hidden gates. This "simulation" allows us to have fewer

LOTOS operators to consider (thus more generality) for the creation, the transformations and the verification of our specifications. Some special symbols in [BuC 93] and [Buh 93a] will not be discussed in depth since they may not be essential to the notation.

Our interpretation of timethreads may often result in a new style of LOTOS code, i.e., with a lot of concurrent instances and many resulting *stop* processes. This *timethread style* reflects the timethread structure of the system under design but not its final architecture. Since we are only concerned with a purely behavioural interpretation without any architectural consideration, at least for the moment, this fact is not a real problem.

3.2 Levels of Specifications

Figure 2 : Basic timethread

Figure 2 represents a *basic timethread*, or a cause-to-effect relationship. It would be easy to think about this behaviour in a sequential way and to define its LOTOS equivalence as `P:= A;` **`stop`**, where *A* represents a sequence of *activities*. A timethread's activity can identify future fragments of sequential code: an abstract sequence of actions, a function, a procedure, a method or parts of processes. Timethread activities are mapped onto LOTOS gates: gates without interaction (from the environment or other timethreads) for actions, and gates on which there is interaction for events (refer to §2.1 for the terminology).

We should also consider the start point and the end point as LOTOS gates. The start point has a triggering event coming from the environment (or from another timethread) and the end point has a resulting event that will be called *Result* when it goes to the environment, and *Continue* when it triggers another timethread. Thus, a unique instance of this timethread could be represented as follows (we deliberately forget the gate parameters for conciseness although they should be all present in each definition and instantiation):

```
process P[...] : noexit :=
   TriggerP; A; ResultP; stop
endproc (* process P, level 1 without recursion *)
```

Nevertheless, since we deal with a reactive system, our timethread's representation must be able to react to more than one initial stimulus from its environment, i.e., we would like this process to be executed as often as the environment desires to. Hence, some recursion has to be included in the process definition:

```
process P[...] : noexit :=
   TriggerP; A; ResultP; P[...]
endproc (* process P, level 1 with recursion *)
```

We also need these instances to execute concurrently, which is not the case in the last definition. LOTOS parallelism needs to be introduced:

```
process P[...] : noexit :=
   TriggerP; (A; ResultP; stop ||| P[...])
endproc (* process P, level 3 *)
```

As expressed, *A* could be an empty sequence of activities, but the timethread still represents the cause-effect relationship between *TriggerP* and *ResultP*. Besides, since the first action, *TriggerP*, is observable (or synchronized with other timethreads, as it will be explained later), unguarded recursion is avoided[1].

For execution purposes, we may prefer not to have an unbounded number of instances of a timethread at once in a system. Hence we could parametrize the maximum number of instances using, for example, the *NumberInstances* abstract data type:

```
type  NumberInstances is NaturalNumber
opns  Pred : Nat -> Nat
eqns
      forall x : Nat
      ofsort Nat
         Pred(Succ(x)) = x;
endtype
```

This parametrized number of concurrent instances could be done, for example, using recursion and selection predicates in the following way:

1. A first attempt in defining this kind of recursion was `P := (A; stop ||| i; P)`, Although recursion is guarded, this process introduces infinite sequences of internal events. This kind of problem must be avoided for the verification and execution of LOTOS specifications.

```
process P[...] (n:Nat): noexit :=
(* n is the maximal number of instances *)
TriggerP; (
   A; ResultP; P[...] (Succ(0))
   |||
   [n ne Succ(0)] -> P[...](Pred(n))
   )
endproc (* process P, level 2 with recursion and *)
        (* with concurrent execution *)
```

The guard *[n ne Succ(0)]* together with the parametrized recursion *P(Pred(n))* instantiate *n* instances of process *P*, as in a countdown, named *P(n)* to *P(Succ(0))*. Then, no other concurrent process will be created. Tail recursion (*P(Succ(0))*) will keep the number of instances to *n* in the system.

Another possibility would be to instantiate an absolute maximum of *n* occurrence of process *P* in parallel, without any tail recursion. Therefore, only *n* concurrent instances will exist and terminate:

```
process P[...] (n:Nat): noexit :=
(* n is the maximal number of instances *)
TriggerP; (
   A; ResultP; stop
   |||
   [n ne Succ(0)] -> P[...](Pred(n))
   )
endproc (* process P, level 2 without recursion and *)
        (* with concurrent execution *)
```

The last possibility presented here is a parametrization where we have a bounded number (*n*) of instances, executed sequentialy:

```
process P[...] (n:Nat): noexit :=
(* n is the maximal number of instances *)
TriggerP; (
   A; ResultP;
      ( [n ne Succ(0)] -> P[...](Pred(n)) )
   )
endproc (* process P, level 2 with sequential *)
        (* execution *)
```

Many different types of behaviours could be associated with one timethread. Depending on what exact behaviour we want to simulate or verify, some levels of abstractions, with their options, can be defined.

Table 1 presents a summary of options associated to our levels of specification. A short example (without gate parameters) is given for each:

Table 1: Levels of abstraction and their options.

Level	*Options*	*Example*

L1: Single instance	Without tail recursion	`process P : noexit :=`
		`  TriggerP; A; ResultP; stop`
		`endproc`

	With tail recursion	`process P : noexit :=`
		`  TriggerP; A; ResultP; P`
		`endproc`

L2: Parametrized number of instances	With sequential execution	`process P (n:nat) : noexit :=`
		`  TriggerP;(`
		`  A; ResultP;`
		`  ( [n ne Succ(0)] -> P(Pred(n)) ))`
		`endproc`

	Without tail recursion, Concurrent execution	`process P (n:nat) : noexit :=`
		`  TriggerP;(`
		`  A; ResultP; stop`
		`  |||`
		`  [n ne Succ(0)] -> P(Pred(n)) )`
		`endproc`

	With tail recursion, Concurrent execution	`process P (n:nat) : noexit :=`
		`  TriggerP;(`
		`  A; ResultP; P(Succ(0))`
		`  |||`
		`  [n ne Succ(0)] -> P(Pred(n)) )`
		`endproc`

L3: Unbounded number of instances	None	`process P : noexit :=`
		`  TriggerP;(`
		`  A; ResultP; stop`
		`  |||`
		`  P )`
		`endproc`

Depending on what questions we want to ask to a generated specification, and on how much detail we want to consider, we may prefer to use different levels. For example, if we wish to quickly test some behaviours or *play* some easy scenarios, a level 1 (L1) specification should be enough and could be rapidly generated and tested. For more complex and realistic scenarios or for the generation of tests for the implementation, a level 2 specification could be used. The last level (L3) may be useful to check equivalences between a

refined specification and a previous one, to validate some kind of extension between a system with added behaviours and the previous system, or to cover a level 2 specification without committing to a specific number. Note that (L3) has a semantics nearly equivalent to the Petri nets' presented in [FCB 93], while (L1) gives more workable and understandable but less complete LOTOS code.

Of course, a natural extension of this concept would be to allow *mixed-levels specifications*, i.e., each timethread would independently have its own level and options. These specifications could simulate the behaviour of a final system in a very realistic way and would be more implementation-oriented than pure L3, L2 or L1.

Using such concurrent and recursive interpretation, the execution of our specification will result in a large number of *stop* processes interleaving with the rest of the behaviour. Although this type of resulting behaviour is usually unwanted, it does not really lead to any problem, even for simulation tools (like XELUDO or LITE). What is really dangerous is the recursion in parallelism (levels 2 and 3), which is not accepted by some tools (for instance, the tool CAESAR).

One way to avoid problems arising from recursion in parallelism might be to add a macro command in a metalanguage (or a tool control language) to manage the number of instances of a process. No option would be needed with such an operator: level 3 specifications could always be used for any simulation. Further research is needed to solve this issue.

3.3 Instance Identifier

For verification and simulation purpose, we may like to formalize the separate identity of timethread instances. Each instance has an implicit state or a set of local variables that determine, e.g., choices of (guarded) paths taken along it. Observe figure 3:

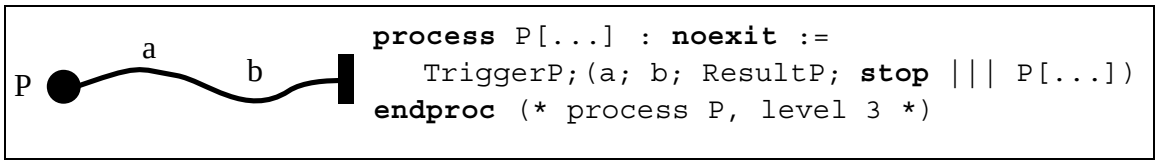

Figure 3 : Timethread without instance identifier

What happens when the environment is:

```
TriggerP; TriggerP; a; ...
```

The first two events will instantiate two concurrent processes and the activity *a* will have to synchronize with one or the other, non-deterministically (see fig. 4). One may raise the question if we need this kind of verification or if we want the first action *a* to synchronize with the first instance, the second action *a* with the second instance, etc.

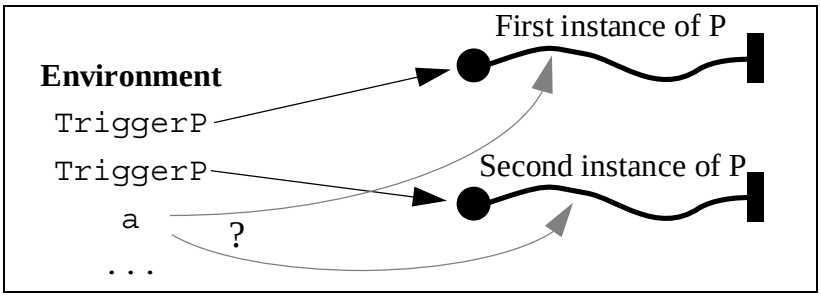

Figure 4 : Non-determinism on activity *a*

At a high level of abstraction, we shall not need to determine which timethread we are currently dealing with. We have to let this situation as abstract as possible. If a distinction is really needed, the designer could include such identification mechanism or we could provide an *instance identifier* to timethreads and events. This identifier could be a natural number associated to (some) events and actions in a timethread:

```
process P[...] (id: Nat) : noexit :=
   TriggerP!id ;(a!id ; b!id ; ResultP!id; stop
                |||
                P[...](Succ(id)) )
endproc (* process P, level 3 *)
```

This approach has however many weaknesses:

- How will the environment associate identifiers to its events? How can we be sure an event goes to a specific timethread while avoiding possible unwanted deadlocks?
- How do we represent, verify and test a race problem in such a specification?
- Isn't the identification of some activities part of a kind of protocol to be designed anyway?

This problem will not be discussed any further in this document, although it will have to be solved at a later stage.

4 Single Timethreads in LOTOS

We now look at some of the basic elements of the Timethread notation in order to define some correspondence of its semantics in LOTOS. A true semantics (level 3 in §3.2) will be always given first, and a few timethreads will also have a level 1 and/or level 2 description[1], to give a general idea of what these levels could look like. Some problems and restrictions will be raised and discussed. Note that the full LOTOS syntax (gate parameters and process identification) is not respected here.

Section 4.1 presents basic timethread combinations, i.e. unconstrained and constrained starts, and the loop constructor. Section 4.2 shows the use of concurrent and alternate segments within a given timethread.

4.1 Basic Combinations

Sequence

Figure 5 : Basic timethread, unconstrained start

The *basic timethread* has been already fully discussed in §3.2. Therefore, here are the three specifications given in §3.2:

```
P := TriggerP; A; ResultP; P (*L1*)

P(n:Nat) := TriggerP; (A; ResultP; P(Succ(0)) (*L2*)
                       |||
                       [n ne Succ(0)] -> P(Pred(n)) )

P := TriggerP; (A; ResultP; stop ||| P)  (*L3*)
```

Constrained Start

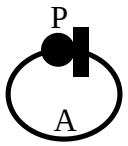

Figure 6 : Constrained start

The corresponding behaviour of figure 6 is that a constrained start timethread only accepts one instance of process *P* at a time in the system, i.e., *P* has to terminate for a new instance to start. However, the triggering event should not be refused, for level 2 and level 3 specifications, while an instance is executed. In fact, those triggering events have to be accumulated to allow many instances of *P* to be executed, one at a time.

1. We will use the tail recursion option of L1 and L2, and also the concurrent option of L2. Refer to §3.2 for more details.

This representation of a constrained start is a shortcut notation for this timethread:

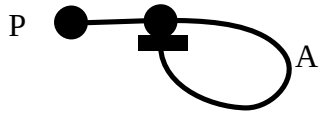

Figure 7 : Equivalent behaviour of a constrained start

Thus, we obtain the following LOTOS representations:

```
P := TriggerP; A; ResultP; P  (*L1*)


hide Sync in P |[Sync]| P2  (*L2*)
where
   P(n:Nat) := TriggerP; (Sync; P(Succ(0))
                          |||
                          [n ne Succ(0)] -> P(Pred(n)))
   P2 := Sync; A; ResultP; P2


hide Sync in P |[Sync]| P2  (*L3*)
where
   P := TriggerP; (Sync; stop ||| P)
   P2 := Sync; A; ResultP; P2
```

This behaviour possesses a representation very similar, w.r.t synchronization on extra hidden gates, to those of §5.3.

Loop

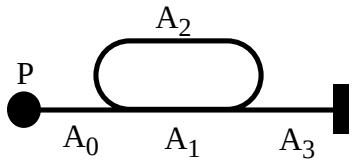

Figure 8 : Loop

In the LOTOS representation of the general loop, we have to define a sub-process (*Psub*) corresponding to the loop part and the ending part. Of course, recursion still has to be supported as well for levels 2 and 3:

```
P := TriggerP; A0; Psub
where
   Psub := A1; (A2; Psub [] A3; ResultP; P) (*L1*)
```

```
P(n:Nat) := TriggerP; (A0; Psub                (*L2*)
                        |||
                        [n ne Succ(0)] -> P(Pred(n)) )
where
   Psub := A1; (A2; Psub [] A3; ResultP; P(Succ(0)) )


P := TriggerP; (A0; Psub ||| P)
where
   Psub := A1; (A2; Psub [] A3; ResultP; stop) (*L3*)
```

Note that A_1, A_2 and A_3 can be empty sequences, as long as A_1 and A_2 are not both empty at the same time. The latter case would not be a desirable LOTOS expression for execution, although it would be a valid one from a syntactic point of view. We already said that a timethread is never really empty, but an empty loop (without any event nor activity) could be a sign of a bad design.

4.2 Concurrent and Alternate Segments

Two very intuitive junction points (OR paths and AND paths) will be used to represent concurrent and alternate segments. Each can have 2, 3, 4 or more branches. Two examples or fork-join with 3 branches are presented to give a general idea of the LOTOS interpretation of these behaviours.

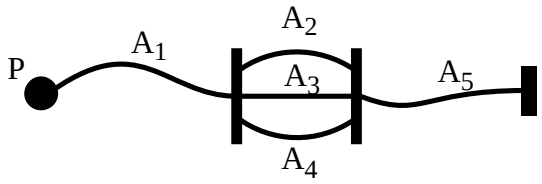

Figure 9 : Fork-join (concurrent segments with an AND junction point)

In the fork-join path presented in figure 9, A_2, A_3 and A_4 may represent a sequence of more than 1 activity. In a previous attempt to define this behaviour, we were using the ***exit*** and enable operator (>>) to synchronize concurrent segment. Here, we prefer to stay more homogeneous and consistent by using the hiding operator with a synchronization gate:

```
P := hide Sync in
      TriggerP;((A1; ( A2; Sync; stop
                     |[Sync]|
                     A3; Sync; stop
                     |[Sync]|
                     A4; Sync; stop )
                |[Sync]| (Sync; A5; ResultP; stop))
      ||| P)   (*L3*)
```

The same type of structure applies when we start concurrent threads (§5.2).

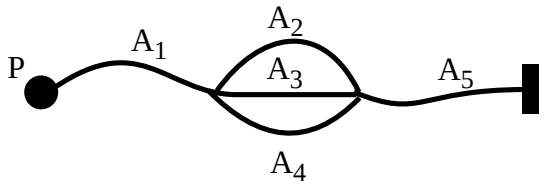

Figure 10 : Fork-join (alternate segments with an OR junction point)

In the alternate segments of figure 10, one and only one path (A_2, A_3 or A_4) will be executed:

```
P := hide Sync in
      TriggerP;((A1; ( A2; Sync; stop
                     []
                     A3; Sync; stop
                     []
                     A4; Sync; stop )
                |[Sync]| (Sync; A5; ResultP; stop))
      ||| P)   (*L3*)
```

In the refinement process, conditions or predicates will have to be somehow inserted to determine which guarded path to follow, but this will not happen until we consider data as part of our model.

In [BuC 93], the difference between *xor* (exclusive or) and *ior* (inclusive or) for alternate segments of a timethread is explained. We will probably formalize only the *xor* part since this one is easier to define and probably more useful for designers.

5 Simple Timethreads Interactions

A timethread diagram of a system is viewed as a collection of interacting timethreads. We can differentiate two types of interactions: starts (§5.2), where one or many timethreads start one or many other timethreads, and other synchronous and asynchronous interactions (§5.3), where different interactions occur along timethread paths. The structural part of these interactions will be obtained using a LOTOS structural method defined in [Bor 93] (§5.1).

5.1 LOTOS Architectural Representation Graphs

In [Bor 93], a LOTOS architectural interpretation method is defined. This method is based on a particular type of architectural interpretations called LARG (LOTOS Architectural Representation Graph) from which architectural expressions are generated. These graphs are very useful to timethread diagrams because interaction architectures are easily obtained in a LOTOS format, and timethreads' local behaviours can then be expressed following the approach presented in section 4.

Figure 11 shows an example of the LARG representation of a small timethread diagram:

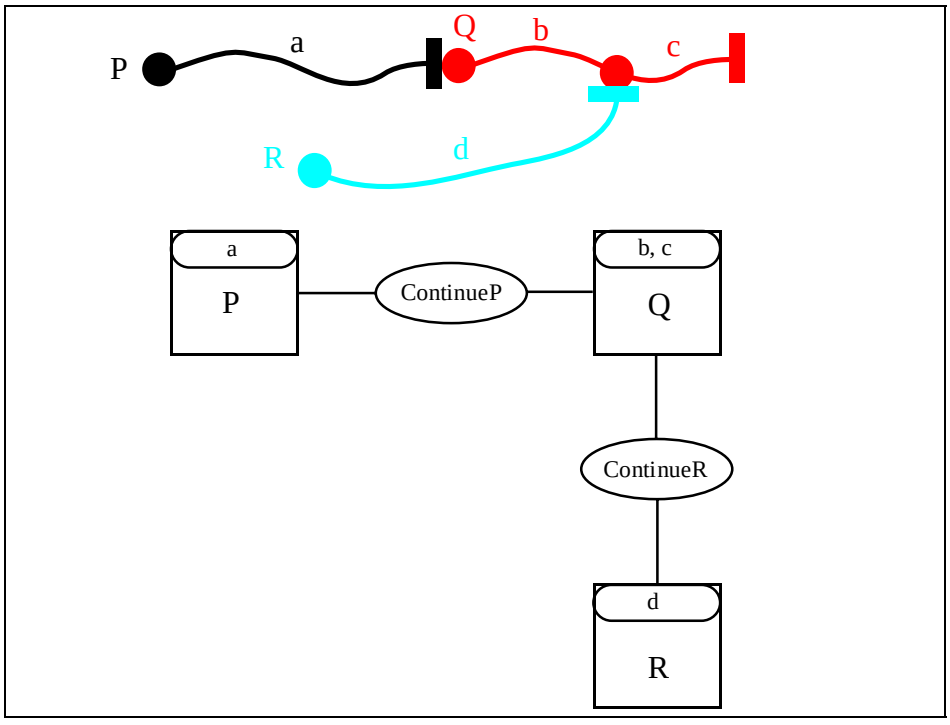

Figure 11 : Example of a timethread diagram and its LARG representation

The LOTOS architecture of such a LARG (binary grouped, however) would simply be:

```
(P |[ContinueP]| Q) |[ContinueR]| R
where ... (* behaviour of processes P, Q and R *)
```

By considering timethreads as entities or processes in their own rights, without reference to the system architecture, modularity is achieved and behaviours (and constraints) can be described locally on a timethread by timethread basis.

Thus, to complete our example, the behaviours of *P*, *Q* and *R* have to include the pertinent gates at the right place. Such behaviours are presented in §5.2 and §5.3[1]. Note that in the next two sections, synchronization gates are *hidden*, i.e., not accessible from the environment. On a timethread design, what is to be shown to the environment and what is to be hidden is still a research topic. Since we do not consider any skeleton architecture to present what should be hidden, we will not propose any solution to this problem right now. However, we will consider any combination of hidden and accessible gates valid for the moment.

5.2 Starting Concurrent Timethreads

In this section, we present many different scenarios where timethreads are started.

Off End Start

Figure 12 : Off end start

Here is one of the most simple interaction: the off end start. *P* will enable *Q* when it terminates and *Q* cannot start by itself. This could represent a sequence that has been refined for some design decision. The corresponding LOTOS behaviour is:

```
(* Timethreads' architecture section *)
hide ContinueP in P |[ContinueP]| Q  (*L3*)
where
   (* Timethreads' behaviours section *)
   P := TriggerP; (A0; ContinueP; stop ||| P)
   Q := ContinueP; (A1; ResultQ; stop ||| Q)
```

Again, we see that guarded recursion is present in both processes. Some precisions are needed here: we do not have an event *TriggerQ* any more because the triggering event of timethread *Q* is now *ContinueP* from timethread *P*.

1. Examples of LOTOS architectures will also be present, although LARGs will not be shown.

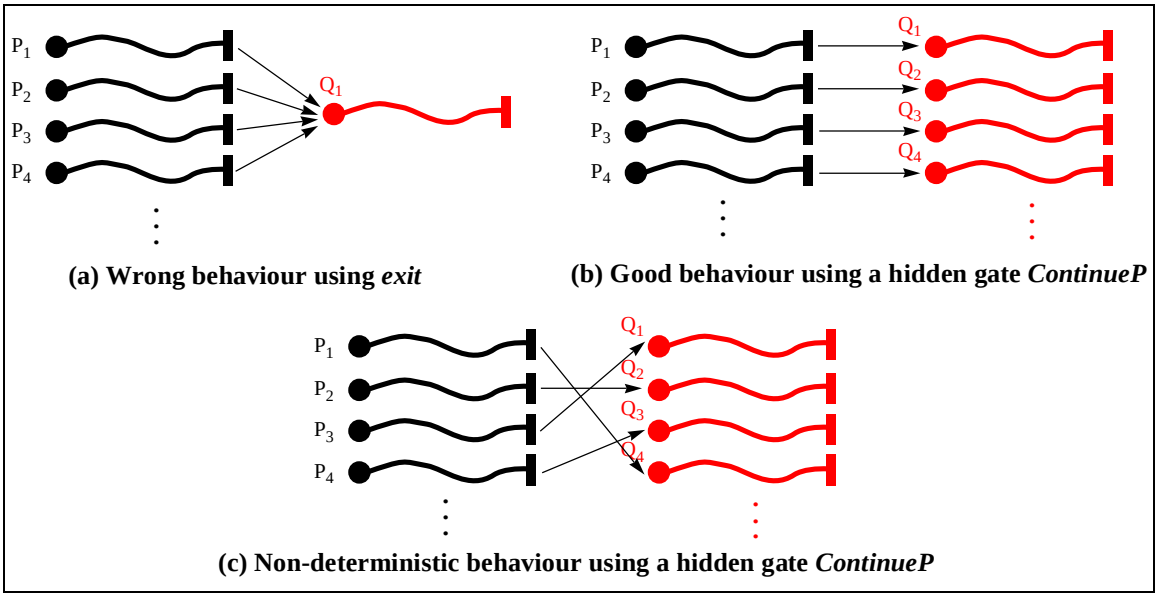

Figure 13 : LOTOS synchronization on an off end start

With a level 3 semantics, we cannot use the LOTOS enable operator (>>) to describe the internal interaction because all instances of *P*, although they are executed in parallel, would have to synchronize on ***exit*** before *Q* starts (see fig. 13a), which is impossible. Using a hidden gate *ContinueP*, an instance of process *Q* is created every time an instance of process *P* terminates (fig. 13b). Note that the termination order of process *P* instances is not know. For example, as shown in figure 13c, the third instance may terminate first, followed by P_2, P_4 and P_1, and instances of *Q* would be created accordingly[1]. Therefore, figure 13b is a special (or constrained) case of figure 13c, which is the intended and more general behaviour.

Generalized termination, as presented in [QuA 92], would be an other option to represent this behaviour. In their document, the authors present an unsynchronized termination obtained by a new ***exit*** process and a new enabling operator ($>e_i>$), which could solve the ***exit*** problem of figure 13a. Nevertheless, as explained in §3.1, adding hidden gates for timethreads synchronization is more homogeneous and consistent with our work than using ***exit*** and the generalized enable operator. Moreover, the extension of [QuA 92] needs some changes in the underlying model of LOTOS (a new compound event) and their proposal is far from being standardized.

A level 1 specification of figure 12 looks like this:

1. This situation is not a real problem since all instances of process *Q* are the same when created. Data from an instance of *P* may eventually be passed to an instance of *Q* through gate *ContinueP* when our model will support data flow.

```
hide ContinueP in P |[ContinueP]| Q   (*L1*)
where
   P := TriggerP; A0; ContinueP; P
   Q := ContinueP; A1; ResultQ; Q
```

If *P* and *Q* are the refinement of a previous timethread, then the refinement would be similar to a pipe or a production line; as soon as *P* terminates, *Q* can start while *P* can work on something else. At a level 1, the previous unrefined timethread (P_{prev}) could be represented by figure 14:

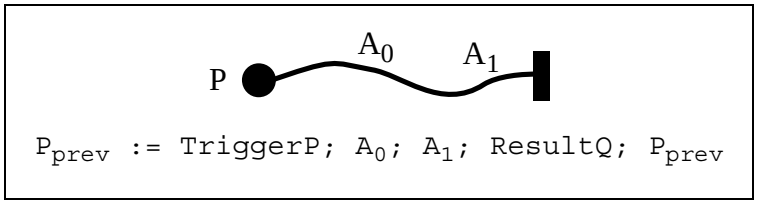

Figure 14 : Timethread before refinement of figure 12

Interestingly enough, we can now observe that P_{prev} and the composition of *P* and *Q* are not weak bisimulation equivalent with level 1 specifications since there is this production line effect. Note however that a level 3 off end start is weak bisimulation equivalent to its level 3 unrefined basic timethread. This kind of relations has to be considered while studying possible transformations.

In Passing Start

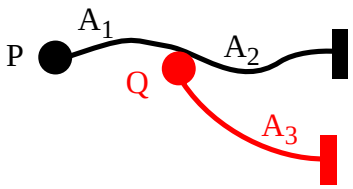

Figure 15 : In passing start

```
hide Sync in P |[Sync]| Q   (*L3*)
where
   P := TriggerP; (A1; Sync; A2; ResultP; stop ||| P)
   Q := Sync; (A3; ResultQ; stop ||| Q)
```

In the "in passing" start of figure 15, timethread *P* never really waits since there is always a process *Q* ready to synchronize with *P*.

The last four figures of this section present other types of scenarios where timethread instances are created. Only final results are given since they do not really need additional explanations.

Or (Fork) Start

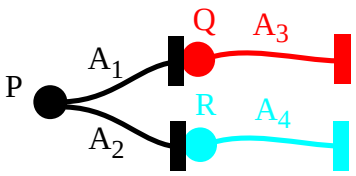

Figure 16 : Or (fork) start

```
hide ContinuePQ, ContinuePR in   (*L3*)
   P |[ContinuePQ, ContinuePR]| (Q ||| R)
where
   P := TriggerP;((A1; ContinuePQ; stop
                  []
                  A2; ContinuePR; stop)
                 ||| P)
   Q := ContinuePQ; (A3; ResultQ; stop ||| Q)
   R := ContinuePR; (A4; ResultR; stop ||| R)
```

Or (Join) Start

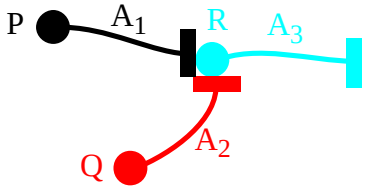

Figure 17 : Or (join) start

```
hide ContinueR in (P ||| Q) |[ContinueR]| R   (*L3*)
where
   P := TriggerP; (A1; ContinueR; stop ||| P)
   Q := TriggerQ; (A2; ContinueR; stop ||| Q)
   R := ContinueR; (A3; ResultR; stop ||| R)
```

And (Fork) Start

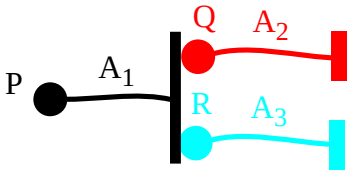

Figure 18 : And (fork) start

```
hide ContinueQR in
   P |[ContinueQR]| Q |[ContinueQR]| R   (*L3*)
where
   P := TriggerP; (A1; ContinueQR; stop ||| P)
   Q := ContinueQR; (A2; ResultQ; stop ||| Q)
   R := ContinueQR; (A3; ResultR; stop ||| R)
```

And (Join) Start

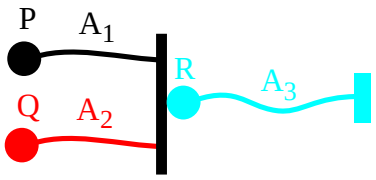

Figure 19 : And (join) start

```
hide ContinueR in
   P |[ContinueR]| Q |[ContinueR]| R   (*L3*)
where
   P := TriggerP; (A1; ContinueR; stop ||| P)
   Q := TriggerQ; (A2; ContinueR; stop ||| Q)
   R := ContinueR; (A3; ResultR; stop ||| R)
```

The hide operator and synchronization on on events are a powerful combination letting us specify these types of behaviours quite easily in a homogeneous way. The next section shows more about the usefulness of this LOTOS operator.

5.3 Other Synchronous and Asynchronous Interactions

This section deals with different kinds of shared paths, synchronizations and triggering events between timethreads. Waiting places on the timethreads will be used. The quite exhaustive list of behaviours may appear unnecessary; it is however there to show that almost any combination can be easily expressed in LOTOS following the same approach.

Join-Fork (Synchronous)

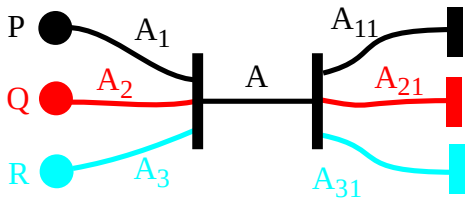

Figure 20 : Join-fork (synchronous)

One easy way to formalize the synchronous join-fork would be the following representation where the synchronization between processes *P*, *Q* and *R* is on all gates included in the sequence *A*:

```
P |[a0,...an]| Q |[a0,...an]| R   (*L3*)
where
   P := TriggerP; (A1; A; A11; ResultP; stop ||| P)
   Q := TriggerQ; (A2; A; A21; ResultQ; stop ||| Q)
   R := TriggerR; (A3; A; A31; ResultR; stop ||| R)
```

However, this synchronization on observable gates is not very meaningful in a timethread context. Moreover, synchronization in all other timethread combination is done on hidden gates. Therefore, for homogeneity purpose, we introduce a new process, executing the sequence *A*, and we synchronize all these processes on hidden gates. This method will also ease transformations like regrouping and splitting:

```
hide S1, S2 in
   SynchroPQR
   |[S1, S2]|
   P |[S1, S2]| Q |[S1, S2]| R   (*L3*)
where
   P := TriggerP; (A1; S1, S2; A11; ResultP; stop ||| P)
   Q := TriggerQ; (A2; S1, S2; A21; ResultQ; stop ||| Q)
   R := TriggerR; (A3; S1, S2; A31; ResultR; stop ||| R)
   SynchroPQR := S1; (A; S2; stop ||| SynchroPQR)
```

This solution corresponds to the usual way of representing a semaphore in LOTOS.

Of course, the data that could be passed from *P*, *Q* and *R* to *SynchroPQR* and then passed back to *P*, *Q* and *R* will require some special attention. However, we are not concerned with data for the moment.

Join-Fork (Asynchronous)

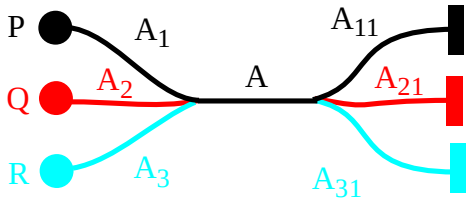

Figure 21 : Join-fork (asynchronous)

```
P ||| Q ||| R   (*L3*)
where
   P := TriggerP; (A1; A; A11; ResultP; stop ||| P)
   Q := TriggerQ; (A2; A; A21; ResultQ; stop ||| Q)
   R := TriggerR; (A3; A; A31; ResultR; stop ||| R)
```

The join-fork of figure 21 represents simply 3 parallel timethreads sharing common activities (*A*) without any synchronization. This is probably the easiest way to represent this behaviour.

End-Trigger

The next figure represents a process *P* waiting (after the execution of sequence A_1) for an instance of *Q* to terminate before continuing:

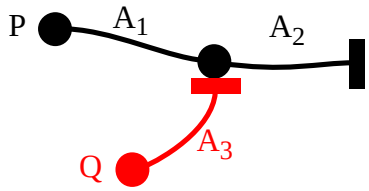

Figure 22 : End-trigger

```
hide ContinueP in P |[ContinueP]| Q   (*L3*)
where
   P := TriggerP; (A1; ContinueP; A2; ResultP; stop
                   |||
                   P)
   Q := TriggerQ; (A3; ContinueP; stop ||| Q)
```

Trigger-In-Passing

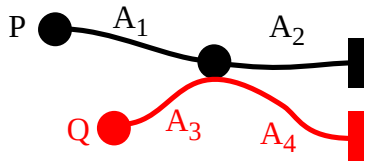

Figure 23 : Trigger-in-passing

The synchronization part or the diagram presented in figure 23 is really "asynchronous" for *Q*, i.e., the timethread *Q* will never wait for *P* while *P* has to wait for *Q* to be ready:

```
hide ContinueP in P |[ContinueP]| Q   (*L3*)
where
   P := TriggerP; (A1; ContinueP; A2; ResultP; stop
                   |||
                   P)
   Q := TriggerQ; (A3; (ContinueP; stop
                        |||
                        A4; ResultQ; stop)
                   |||
                   Q)
```

The last three figures show some combinations, with more than 2 timethreads acting in the synchronization, of the previous behaviours:

And-Trigger

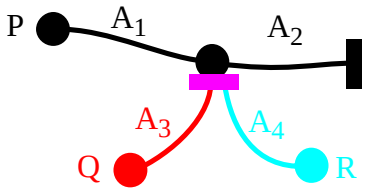

Figure 24 : And-trigger

```
hide ContinueP in
   P |[ContinueP]| Q |[ContinueP]| R   (*L3*)
where
   P := TriggerP; (A1; ContinueP; A2; ResultP; stop
                   |||
                   P)
   Q := TriggerQ; (A3; ContinueP; stop ||| Q)
   R := TriggerR; (A4; ContinueP; stop ||| R)
```

And-Trigger (1 In-Passing)

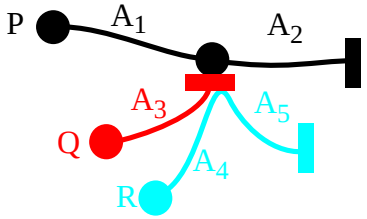

Figure 25 : And- trigger (1 in-passing)

```
hide ContinueP in
   P |[ContinueP]| Q |[ContinueP]| R   (*L3*)
where
   P := TriggerP; (A1; ContinueP; A2; ResultP; stop
                   |||
                   P)
   Q := TriggerQ; (A3; ContinueP; stop ||| Q)
   R := TriggerR; (A4; (ContinueP; stop
                        |||
                        A5; ResultR; stop)
                   |||
                   R)
```

Or-Trigger

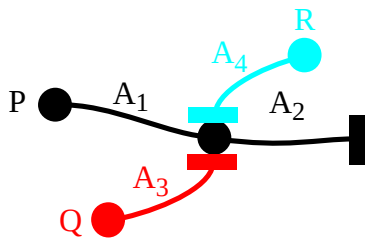

Figure 26 : Or-trigger

```
hide ContinueP in
   P |[ContinueP]| (Q ||| R)   (*L3*)
where
   P := TriggerP; (A1; ContinueP; A2; ResultP; stop
                   |||
                   P)
   Q := TriggerQ; (A3; ContinueP; stop ||| Q)
   R := TriggerR; (A4; ContinueP; stop ||| R)
```

More complex behaviours can also be described using the same approach. A separate architectural approach, like the LARG method presented in [Bor 93], combined with a timethread-by-timethread description, gives us the LOTOS architecture required to represent the most complex timethreads interactions. The method for such complete timethread diagrams is presented in [BoA 93].

6 Special symbols

We discuss here some special-purpose timethread symbols mentioned in the literature [BuC 93] [Buh 93a].

6.1 Timer

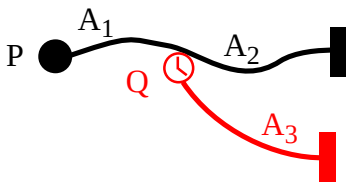

Figure 27 : After timeout (or delayed) start

```
hide Sync in P |[Sync]| Q   (*L3*)
where
   P := TriggerP; (A1; Sync; A2; ResultP; stop ||| P)
   Q := hide TimeOut in
            Sync; (TimeOut; A3; ResultQ; stop ||| Q)
```

The timer symbol is a waiting place which is triggered by a timeout. In fig. 27, the internal action *TimeOut* in timethread *Q* represents the desired delay or timeout. Note here that the timeout represents effectively a delay, because *TimeOut* will always occur.

Time is a very abstract notion in LOTOS and we will have to determine what timed extensions (if any) would be the most appropriate to us to represent this behaviour.

6.2 Stubs

Stubs represent a non-refined behaviour. They are treated as activities representing part of a process that has to be defined at a later stage (fig. 28a), or as empty timethreads when they are not directly on the body of another timethread (fig. 28b). Stubs could then be refined easily while providing a better semantics sooner.

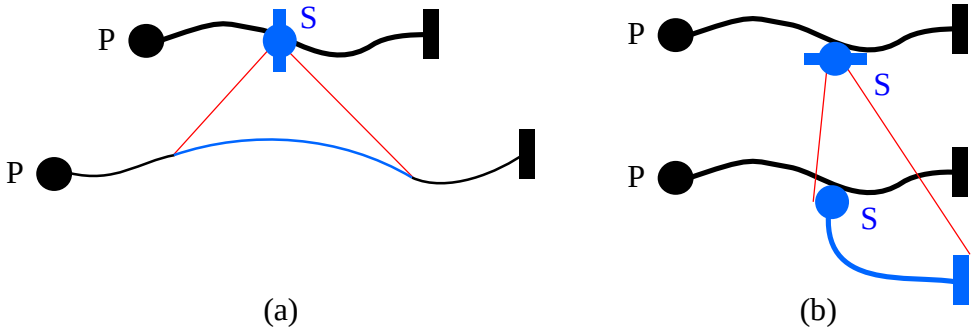

Figure 28 : Possible LOTOS refinement for stub *S* in timethread *P*

Discontinuities are another type of constructor that are very similar to stubs. Since they are not essential to us (at least for the moment), we will not consider them as part of our basic timethread set.

6.3 Abort

This operator has to be associated with the LOTOS disabling operator ([>). Note that an abort really kills all instances of a timethread until a new instance is triggered. Figure 29 shows an example of a timethread *Q* aborting a timethread *P*:

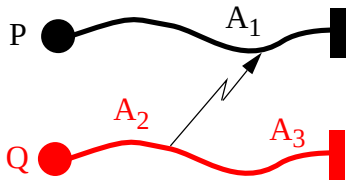

Figure 29 : Example of an abort

```
hide Abort in P |[Abort]| Q    (*L3*)
where
   P := TriggerP; (A1; ResultP; stop ||| P)
                  [> Abort; P
   Q := TriggerQ; (A2; Abort; A3; ResultQ; stop
                   |||
                  Q)
```

Note here that the *Abort* is really hidden, and that all instances of *P* can be killed anytime after they have been triggered.

It is difficult to know if this approach becomes quickly too complex or even impossible to deal with. Nevertheless, some simple examples could be developed to see what are the real problems and limitations of this LOTOS correspondence.

6.4 Other Symbols

Other special symbols, introduced in the timethread literature, will not be fully formalized since their usefulness may not worth their complexity. Memoryless waiting places and different relationships are such symbols. They will not be part of our basic timethread set until we get a better understanding of their exact meaning and importance.

7 Discussion

7.1 Action Refinement

Actions in the basic timethread can represent many different abstract concepts: a procedure, a function, a method, etc., which could be refined later, in a design process, into a more concrete representation. *Action refinement* is recognized as a fundamental technique for designing complex systems, as it permits to consider designs at different levels of abstraction. Note that we use the term *action* because we do not consider *events* yet in our refinement process. In [CoS 93], two ways of performing action refinement are defined: (i) syntactical substitution, i.e., substituting a process for any action to be refined, and (ii) by using a new operator of the language.

Syntactical substitution appears more difficult to perform than the second option, mainly in respect with synchronization mechanisms, particularly the implicit multiway synchronization of LOTOS. However, we do not really have synchronization on actions in our timethread context, as we saw in §4 and §5. Therefore, syntactical substitution could be a more efficient way to introduce action refinement in our methodology. Also, some equivalence relations have been defined for this type of transformation. We will have to look at existing relations like branching bisimulation [vGW 89], which is a relation with the same expressiveness of a weak bisimulation, but which remains preserved under action refinement.

Refinement by Sequence of Actions

We can use the concept of *event refinement* expressed in [CPT 92]. This is a design transformation in which an action of the initial high-level design is replaced by multiple subactions in the resulting design. This refinement allows concurrent activities to overlap in time, which is not possible using single actions due to the atomicity of LOTOS events.

Since there is no parallelism within an action, we will consider action refinement as the replacement of an action by a sequence of sub-actions ($a \rightarrow A$) or by a whole process ($a \rightarrow P$). Therefore, concurrent actions could be refined by *intervals* to overlap, and not just to interleave with each other. This transformation allows us to test and verify some kind of true concurrency in a multiple timethreads diagram. For example, suppose we have two concurrent timethreads *P* and *Q* (fig. 30) that have actions *a* and *b* respectively:

Figure 30 : Unrefined actions

Using simple interleaving in LOTOS, we can only know that *a* occurs before or after *b*, although *a* and *b* may not be atomic and could be executed on a certain interval of time. These could be refined in the following way:

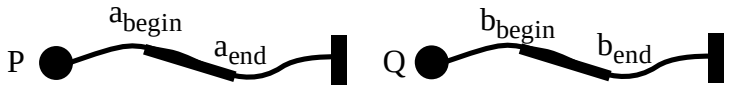

Figure 31 : Refined actions

There could be other intermediate new sub-actions between a_{begin} and a_{end}, and between b_{begin} and b_{end} in figure 31. Although no problem concerning the synchronization of these refined actions will occur[1], their individual hiding will have to be considered.

Refinement by Process

The refinement of an action into a process can be achieved in two ways. The first one is to replace the action by a LOTOS process, e.g., if we want to refine *b* into *P* in the sequence *a; b; c; ...*, then we get *a; P >> c; ...*, where P is a process definition with ***exit*** functionality. The problems here are that we use the enable operator (>>), which may complicate other correctness preserving transformations, and that there is no real correspondence between *P* and its timethread correspondence. Furthermore, due to the semantics of the LOTOS parallel operator (|||), all instances of a process *P* would have to synchronize with each other before *Q* could start (see fig. 13a). The second (and better) option is to use stubs (see §6.2) instead of an action to be refined.

Glue Refinement

Refinement could also be done on the *glue* that links activities on a timethread. Figure 32 shows the link between *a* and *b*. This glue could represent some means by which *a* communicates with *b*.

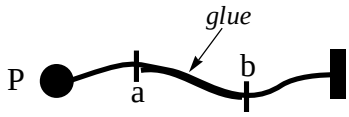

Figure 32 : Glue linking activities

Our LOTOS representation should not consider the glue specification unless it is refined explicitly with other activities, or with a new timethread. A LOTOS transformation cannot guess what is not specified on the timethread! Therefore, we will not consider this glue any further in our approach.

1. *a* and *b* are actions, therefore they will never be part of synchronization gates between two or more LOTOS processes representing timethreads.

Naturally, in our representation, some refinements could be impossible or not meaningful w.r.t. possible equivalence relations. "Given a well-established equivalence notion which is not preserved under refinement, is there a way of restricting either the allowed refinements or the class of system representations under consideration such that preservation of this equivalence in the restricted setting is obtained?" [CGG 92]. Therefore, we will have to work not only on new possible relations for our framework, but also on restrictions on action refinement.

7.2 Context and Constrained Use of Timethreads

In this document, we try to map timethreads diagrams onto LOTOS processes. Since LOTOS can be very (and sometimes too) abstract and can express only some of the timethread's informal semantics, mapping a LOTOS process back onto timethreads could be too complex for us right now. This is true even if LOTOS processes are obtained from a previous mapping. By using formalism, we lose the shape, the environment, the *context* of the timethread, and thus part of its semantics. We may have a *projection* of the timethread design onto LOTOS semantics, but we cannot directly map LOTOS back onto timethreads if we do not remember or consider the context.

If we work with an *informal use* of our Timethread notation, we may be able to map some of its semantics on a LOTOS specification. The latter could however have some inconsistencies (w.r.t. gate names, mapping of timethread constructs, etc.) and many difficulties will arise when working in such a design environment. Since timethreads are abstract, ambiguous and often incomplete from a formal methods point of view, a *constrained use* has to be described. This constrained use has to be more suitable for formal methods (and thus LOTOS) while still being intuitive and attractive for designers.

A constrained use also means that we need a *formal definition* of timethreads diagrams. This strict definition might be represented as a grammar where all information needed by formal methods[1] concerning the timethreads' behaviours will be included. Timethreads interactions might be represented by LARGs [Bor 93], for instance. A single model to represent both timethreads behaviours and interactions might be difficult to find.

Timethreads are a visual notation, thus visual and spatial information has to be recorded somewhere. Our formal definition will probably not include such information, although a more general description, called *intermediate representation*, including both the formal definition and the visual information should also be defined.

1. LOTOS is not the only formal methods to be considered. This description has to be abstract and not focused on a specific Formal Description Technique only. It may be mapped later on other formal methods, e.g., Petri nets or event structures.

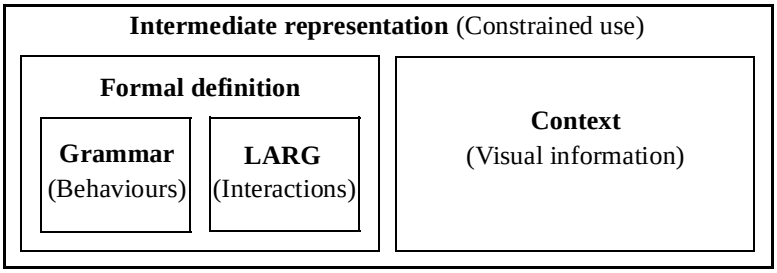

Figure 33 : Intermediate representation of a timethreads diagram

This *intermediate representation* of a timethreads diagram (fig. 33) will be especially useful for interactive design and for automatic transformations at the formal methods level (see §7.3).

In future work, we will have to find from the context what is to be mapped onto LOTOS and what is to be left out exactly. To decide what is to be represented, questions to be asked about the specification have to be defined. Building some examples will help us finding what are the real needs for verification and testing. This will help us defining a useful, pertinent and efficient formal definition.

7.3 Timethread Refinement and LOTOS Relations

One of the timethreads' strengths is the way they can be refined, transformed and extended to represent more detailed and complete behaviours. LOTOS possesses similar characteristics; a process can be refined and extended in many ways. Using correctness preserving transformations [CPT 92] and other LOTOS relations [BoB 87] [KhB 92] could help us formalizing timethread refinement.

When we add new timethreads, activities or alternatives, we may want to preserve some characteristics or some behaviour of our previously specified system. The papers mentioned previously give us some tools and relations already existing to answer such questions:

- Functionality extension,
- Reduction (**red**), conformance (**conf**), extension (**ext**) and other new extension relations,
- Event refinement, etc.

During the mapping of timethreads on a skeleton architecture, we would like our specification to be arranged, composed or decomposed in some specific way (network of components) while preserving the correctness of the defined behaviour. Some methods also exist to solve part of this problem:

- Splitting and regrouping parallel processes [Lan 90],
- Interaction points (gates) rearrangement,

- Bisimulation (strong, weak and congruence), testing and trace equivalences,
- Inverse expansion (to get more parallelism),
- Multiway to two-way synchronization, etc.

General restructuring algorithms for LARGs [Bor 93] are also needed for timethreads splitting, merging, etc.

In the literature, we find a large number of existing correctness preserving transformations and relations. Our focus should not be exclusively on LOTOS transformations and relations. If we look attentively, many other domains can provide excellent hints and ideas to help us defining our own timethread relations and transformations. For example, many equivalence semantics, from trace to testing to bisimulation equivalences, exist for these domains [vGl 90]:

- Graph domains (process graph, state transition diagram)
- Net domains ((labelled) Petri nets)
- Event structure domains ((labelled) event structure)
- Explicit domains (mathematically coded set of properties)
- Term domains (term in a system description language)
- Projective limit domains (projective limits of series of finite term domains)

Since timethreads are often related to simple cases of such existing domains, it could be worthwhile looking at the work already done and adapting it for the Timethread notation afterwards.

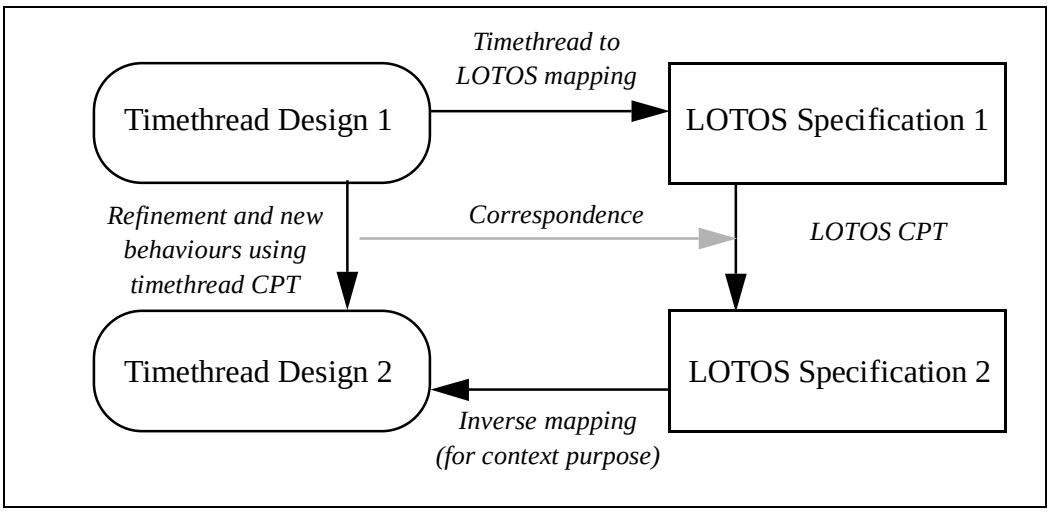

Figure 34 : Refinement using timethread CPT.

Figure 34 presents a refinement approach where timethreads correctness preserving transformations (based on LOTOS CPT) are used. Corresponding LOTOS CPT supporting these timethread transformations in a formal way would have to be defined. Validation is not needed using this type of refinement.

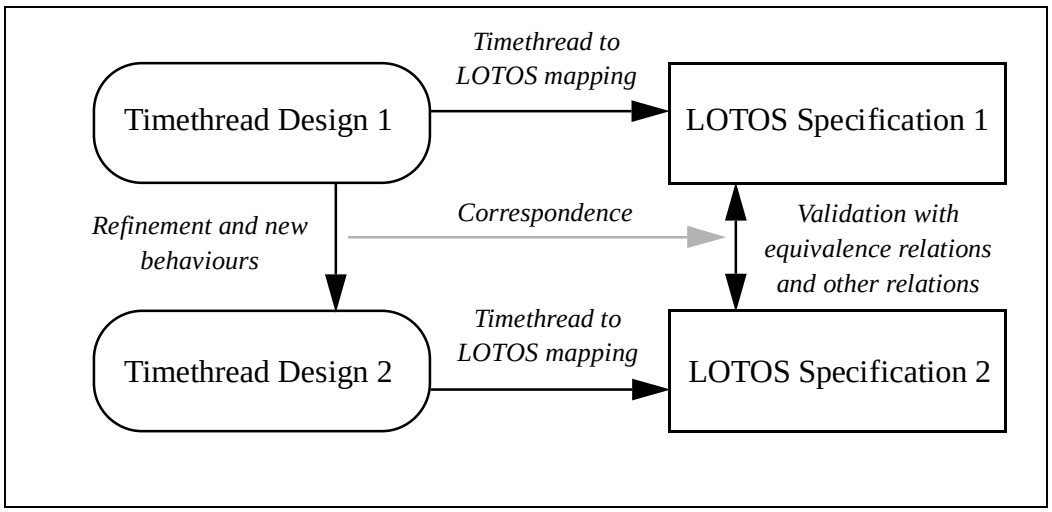

Figure 35 : Refinement using validation.

Where timethreads CPT cannot be defined, a validation approach will have to be taken. Figure 34 illustrates this type of refinement. This situation is however generally not suitable since validation using any known relation is very costly.

Validation is very hard for any but trivial systems, so validation of larger or complex systems is usually not feasible in practice. "A component-based style allows components to be verified individually. Larger combinations (or designs) of trusted components can then be verified more easily" [Tur 93]. In our validation domain, a timethread-based style, timethreads could be validated individually. This validation approach is not obvious for the moment but should be studied further in the future.

In a refined specification, data might be associated to timethreads. LOTOS abstract data types may then have to be used. However, the data part will not be investigated in this document, although we will have to consider its formalization later.

7.4 Design Methodology

Why would we need another design methodology using formal methods while some already exist? For instance, the Lotosphere Methodology [LOT 92], based on the conventional stepwise refinement, offers powerful structuring and abstraction facilities that allow the designer to maintain control of the different aspects of the design at all levels along the design trajectory. This is achieved by enabling formal statements of design constraints and objectives in the structure of the design. The quality of the design is improved because of mathematical foundations of LOTOS, that allow verification of properties and extensive support for testing.

What we need is a thinking tool, a methodology more intuitive and appealing than the one proposed in the Lotosphere project. It has to be used in the framework of a practical design process that engineers in industry can use without having to be formalists. The Lotosphere Methodology takes a formal method (LOTOS) and tries to build a complete design methodology on it, which is not really what we said we needed. In our approach, we start with some visual design concepts, very natural to designers, and we try to use a formal method, LOTOS in our case, to help formalizing some of its part with what LOTOS offers the best. This direction seems to be more promising than the former one.

Although we know how to capture the main requirements with a timethread design and then get the corresponding LOTOS specification (explained in the following sections), this does not mean that we have a complete design framework! We have to get a complete implementation-oriented model, or architectural specification, of our timethread-designed system. Two major approaches are presented here: the *derivation* (fig. 36a) and the *validation* (fig. 36b).

Derivation: In this approach, an architectural specification of the system (***AD***) is obtained from the LOTOS mapping of the timethread design (***TD***) on which correctness preserving transformations like restructuring, process splitting and merging, etc., can be applied.

Validation: Here, we have to design both the timethread diagram and a sketch of an architecture. We then map the two designs onto LOTOS specifications (***TV*** and ***AV***). Finally, we try to validate the LOTOS specification obtained from the architectural design (***AV***), using either some relations or test cases derived from the LOTOS specification obtained from the timethread design (***TV***).

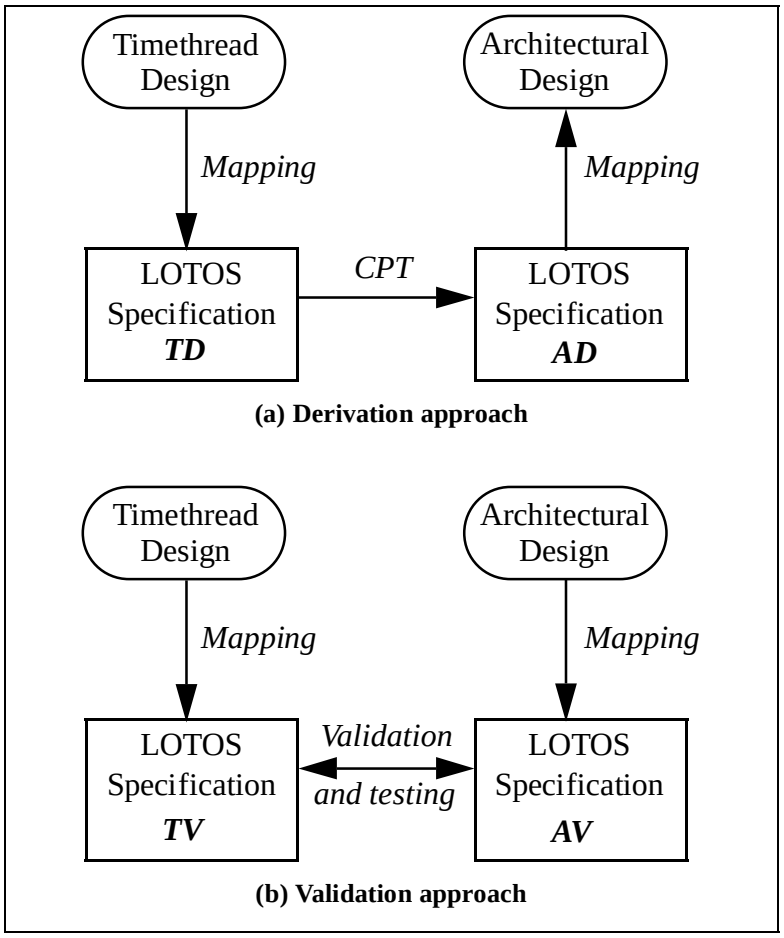

Figure 36 : Derivation and validation approaches

The advantages of using CPT is quite obvious: no need for costly validation or for the generation of test cases. Although this solution is theoretically appealing, CPT are sometime difficult to define and also hard to apply. Nevertheless, this solution seems promising and deserves some good research.

8 Conclusion and Future Work

Although still theoretical, this document gives a good overview of the possibilities of a LOTOS-Timethreads framework. The results of the correspondence of §4 and §5, which describe the LOTOS interpretation of a new design style called *timethread style*, can be very helpful and could lead to a better formalization and use of the Timethread notation as a thinking tool in a design process.

Timethreads seem to be liked by designers, and using LOTOS as an underlying model gives much more power to the notation. Timethreads need to keep their "informal look" to stay appealing for users. However, in order to be used in a comprehensive software development model (from requirements to *validated* implementation), their concepts need to be cast into a more formal framework. This could be LOTOS.

A LOTOS interpretation method for timethreads together with a case study (the traveller system) is presented in [BoA 93]. Other examples of the methodology, like the elevator system, the MTU or a telephony system, will help us defining a complete framework where formal methods are integrated in the design process.

Many topics presented here require further attention. In the short term, we have to:

- Formally define timethread diagrams and the notion of context.
- Define consistent transformations w.r.t. the formal definition of timethreads.
- Look at strengths of the validation and derivation approaches in the design methodology.
- Explore the advantages and drawbacks of a mixed-levels specification of a timethread system.
- Look at what can and cannot be answered by a specification.
- Integrate skeleton architectures in the notation to define what activities are to be hidden from the environment.
- Look at the real necessity of instance identifiers and special symbols.
- Define more clearly constraints to be applied to activity refinement.
- Look at the use of LOTOS CPT for our framework.

In the long term, we could be more ambitious:

- Introduce data in the notation.
- Since we mostly deal with real-time systems, introduction of time concepts will be needed sooner or later. A new underlying model, probably a time-extended LOTOS, should also be studied.
- Define the intermediate representation, more general than what is needed by LOTOS.
- Define the needs for timethread refinement.
- Define Timethreads Correctness Preserving Transformations (TCPT).
- Define equivalence, reduction and extension relations for validation purpose.

- Look at a timethread-by-timethread validation process.
- When we know the limitations of the LOTOS approach w.r.t. validation, testing and verification, other underlying models (e.g., Petri nets or event structures) should be explored to complete the LOTOS approach.
- Define a performance analysis model and metrics applied to timethreads restructuring for performance purpose.
- A real-life case study form industry could be an excellent way to test and improve the design framework, once defined.

Much work remains to be done, but the result could worth the effort.

9 Acknowledgments

Many people, from Carleton, Ottawa and Hull, contributed to this report by their explanations, discussions, comments and encouragements. I am principally grateful to Francis Bordeleau, Professor Ray Buhr and Professor Luigi Logrippo for all the time they passed with me talking about relations between timethreads and LOTOS. Many thanks also to Ron Casselman and Professor Abdellatif Obaid for their comments on early draft versions of this document. Finally, Jacques Sincennes and his team helped a lot with the LOTOS toolkit XELUDO.

This work was funded by several sources: NSERC, FCAR, TRIO and the Ministère de l'Enseignement supérieur et de la Science du Québec.

References

[BoB 87] Tommaso Bolognesi and Ed Brinksma, "Introduction to the ISO Specification Language LOTOS", *Protocol Specification, Testing and Validation VIII* (1988), North-Holland, pp. 23-73

[BCP 93] R.J.A. Buhr, R.S. Casselman and F. Pomerleau, "Timethread-Driven Design of Dual Frameworks for Real-Time and Distributed Systems", TR-SCE-93-06, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1993)

[BDC 92] Luca Bernardinello and Fiorella De Cindio, "A Survey of Basic Net Models and Modular Net Classes", *Lecture Notes in Computer Science #609: Advances in Petri Nets* (1992), Springer-Verlag, pp. 304-351.

[Bor 93] Francis Bordeleau, "Visual Descriptions, Formalisms and the Design Process", Master's thesis, School of Computer Science, TR-SCE-93-35, Carleton University, Ottawa, Canada (1993)

[BoA 93] Francis Bordeleau and Daniel Amyot, "LOTOS Interpretation of Timethreads: A Method and a Case Study", TR-SCE-93-34, Carleton University, Ottawa, Canada (1993)

[BuC 93] R.J.A. Buhr, R.S. Casselman, "Designing with Timethreads", TR-SCE-93-05, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1993)

[Buh 93] R.J.A. Buhr, "Pictures that Play: Design Notations for Real-Time & Distributed Systems", TR-SCE-93-04, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1993)

[Buh 93a] R.J.A. Buhr, "Object Oriented Design of Real-Time & Distributed Systems", Course notes, 94.586, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1993)

[FCB 93] W. Foster, R.S. Casselman, and R.J.A. Buhr, "From Timethreads to Petri Nets: A First Pass", TR-SCE-93-26, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1993)

[CGG 92] Ingo Czaja, Rob J. van Glabbeek and Ursula Goltz, "Interleaving Semantics and Action Refinement with Atomic Choice", *Lecture Notes in Computer Science #609: Advances in Petri Nets* (1992), Springer-Verlag, pp. 89-107.

[CoS 93] J.P. Courtiat and D.E. Saïdouni, "Action Refinement in LOTOS", *Protocol Specification, Testing and Validation XIII* (1993), North-Holland, pp. F3-1 to F3-14.

[CPT 92] Lo/WP1/T1.2/N0045/V03, "Catalogue of LOTOS Correctness Preserving Transformations", T. Bolognesi Editor (1992), Lotosphere Project (ESPRIT 2304)

[GLO 91] S. Gallouzi, L. Logrippo and A. Obaid, "Le LOTOS: Théorie, outils, applications", TR-91-25, Dept. of Computer Science, University of Ottawa, Ottawa, Canada (1991)

[ISO 88] ISO, Information Processing Systems, Open Systems Interconnection, "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", DIS 8807 (September 1987)

[KhB 92] F. Khendek, G. v. Bochmann, "Merging Behavior Specifications", Publication # 856, Département d'IRO, Université de Montréal, Montréal, Canada (1992)

[LaB 92] S. Lamouret and R.J.A. Buhr, "Study of the links between Telos and LOTOS in relation with TimeThreads", Real-Time and Distributed Systems Group, Carleton University, Ottawa, Canada (1992)

[Lan 90] Rom Langerak, "Decomposition of Functionality: a Correctness Preserving LOTOS Transformation", *Protocol Specification, Testing and Validation X* (1990), North-Holland, 229-242

[LOT 92] Lo/WP1/T1.1/N0045/V04, Juan Quemada, Gerard Yadan, "The Lotosphere Design Methodology: Basic Concepts", Luis Ferreirs Pires Editor (1992), Lotosphere Project (ESPRIT 2304)

[QuA 92] Juan Quemada and Arturo Azcorra, "Structuring Protocols using Exceptions in a LOTOS Extension", Technical report, DIT-UPM, Madrid, Spain (1992)

[Roz 92] Brigitte Rozoy, "On Distributed Languages and Models for Concurrency", *Notes in Computer Science #609: Advances in Petri Nets* (1992), Springer-Verlag, pp. 267-291.

[Tur 93] K.J. Turner, "An Engineering Approach to Formal Methods", *Protocol Specification, Testing and Validation XIII* (1993), North-Holland, pp. I3-1 to I3-24.

[vGl 90] R. J. van Glabbeek, "The Linear Time - Branching Time Spectrum", *Lecture Notes in Computer Science #458: CONCUR 90* (1992), Springer-Verlag, pp. 278-297.

[vGW 89] R. J. van Glabbeek and W. P. Weijland, "Refinement in Branching Time Semantics", *AMAST Conference* (1989), Iowa City, USA, pp. 197-201

[ViB 91] Mark Vigder and R.J.A. Buhr, "Using LOTOS in a Design Environment",*Proceeding of FORTE'91, Fourth International Conference on Formal Description Techniques* (1991), North-Holland, pp. 1-14