Formalization of Timethreads Using LOTOS

By

Daniel Amyot

Thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

under the auspices of the
Ottawa-Carleton Institute for Computer Science

University of Ottawa
Ottawa, Ontario, Canada
October 1994

Abstract

Timethreads are a new notation for visual description of the different causality paths of a system. They illustrate causality sequences of activities through systems. A design process based on the use of timethreads has already been defined.

The Formal Description Technique LOTOS (Language Of Temporal Ordering Specification) is a specification language based on the temporal ordering of observational behaviour.

This thesis aims at the integration of formal methods in the design of real-time and distributed systems by presenting a LOTOS interpretation of timethreads. With the help of a timethread grammar and a suite of techniques, LOTOS specifications are derived from timethread maps. The designer can then ‘play’ with the design by validating the specifications during the early stages of requirements capture and analysis.

Acknowledgements

I would like to express my deepest gratitude to my two supervisors, Professor Luigi Logrippo (University of Ottawa) and Professor Raymond J. Buhr (Carleton University). Their continuous support, interesting discussions, and numerous encouragements made my research project most enjoyable.

I am very thankful to my colleague and friend Francis Bordeleau for initiating me to this project and for sharing uncountable hours, ideas, and fruitful comments about this research and this thesis. Many thanks also to Michel Locas for reading previous versions of this work and for making judicious comments about the timethread grammar.

I would like to thank everyone from the LOTOS group of the University of Ottawa for their useful suggestions and patient listening. Also, Jacques Sincennes and his team helped a lot with the tools *ELUDO* and *XELUDO*.

I also owe special thanks to Professor Gerald M. Karam (Carleton University) for providing valuable time, useful information, and several technical reports about the Ontario Telepresence project.

Gracias to Santiago Pavón (Universidad Politécnica de Madrid) for sending a copy of the tool *LOLA* running under Windows.

Merci aussi à Gilbert Babin et Charles Brochu (Université Laval) pour m'avoir fait parvenir leur très utile outil *diaggen* ayant permis de générer les diagrammes syntaxiques.

J'aimerais surtout remercier mon épouse Annie ainsi que mes parents, Jean-Guy et Madeleine, pour leur éternel soutien et leur amour de tous les jours.

Finally, I would like to thank for their financial support: the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche du Québec (FCAR), the University of Ottawa, the Telecommunication Research Institute of Ontario (TRIO), and the Ministère de l'Enseignement Supérieur et de la Science du Québec.

Table of Contents

CHAPTER 1 Introduction

1.1 Motivation

This thesis is part of an ongoing project from the Formal Methods in Design (FMD) research group. This project aims at defining a framework for the integration and support of formal methods in the timethread-centered design process. For that purpose, one of the first steps defined was to show that LOTOS specifications could be generated from timethread designs. This thesis addresses this issue.

Formal Methods in the Design Process

Formal methods in software engineering intend to provide a mathematical foundation to the process of software design, transformation, and validation. Many such methods were developed in the last decade. Their integration in the design process may indeed prove very profitable, if this is done in an appealing and cost-effective way for use in the industrial environment. Nevertheless, industrial developers are not inclined to integrate these new methods in their design processes. Some of the main reasons concern the relative complexity of formal languages, and the new ways of thinking they impose on designers.

A certain degree of informality is essential in early stages of the development process. Designers in industry regularly use a variety of notations in a partially informal manner to capture requirements and candidate solutions. These notations, although informal, are useful as thinking tools. However, one cannot proceed from the informal to the formal by formal means. We cannot go automatically from these informal diagrams and sketches to a complete formal specification. It is also almost impossible to capture the requirements correctly by using formal methods directly. Design decisions have to be made, and many intermediate steps are often required.

We need a less painful way of creating formal specifications in industry. We want a method where designers could use the power of formal techniques through a user-friendly interface. One of the purposes of our work is to be able to capture the requirements and then to do high-level requirement testing in early stages of development. This is very important since the further errors are detected in a product's development, the more costly it is to fix them [Pro 92]: "Fixing a problem in the requirements costs 1% as much as fixing the resulting code" [Pfl 92].

Problem Definition

We believe that formal methods do not intend to replace the whole design process, but their integration in the development process could lead to solutions with fewer errors in a shorter time period. The real question then becomes: *how* should formal methods be integrated in the design process of real-time and distributed systems in an appealing way for industrial engineers?

We think that the integration of formal methods is best achieved when designers do not have to change their way of thinking and communicating. The following requirements, already discussed in [BBO 94], present the main issues from our point of view:

R1) Designers should be allowed to use whichever design description model offers them the expressiveness and flexibility they need to design real-time and distributed systems. The way people actually work does not have to be radically changed.

R2) Designers can use different formal methods to analyze different aspects of these systems. One formal method is not expressive enough to capture the whole design. It is only a projection. A multi-formalism approach has many advantages over a single-formalism approach.

R3) Designers do not have to be experts in a specific formal language to use it. A formal method should be transparent to the user while its strengths are being used.

R4) Since visual notations are more expressive and easier to conceptualize at a high level of abstraction than textual descriptions, they should be used to capture the major concepts and basic scenarios from the requirements.

R5) Tools must be available to help the designer go from the informal to the formal. Tools specialized for a formal method can be used afterwards on the formal description.

Many problems arise from such general requirements. This thesis proposes a solution based on the Timethread notation and the formal description technique LOTOS.

1.2 Objectives

This thesis aims at providing elements for the integration of the formal technique LOTOS in a timethread-centered design process while conforming to the five requirements enumerated in the previous section.

We define four main objectives in this context:

O1) To demonstrate that we can manually generate LOTOS specifications from timethread maps

O2) To show that these specifications are meaningful and that they can be used to execute the design. This is also referred as *play the design*.

O3) To show that tools could eventually support the transformation from timethreads to LOTOS.

O4) To discuss resulting problems, difficulties, and new research issues.

To satisfy these objectives, we will use an approach based on formal interpretation methods. Chapter 3 presents more deeply this approach and the different contributions of the thesis.

1.3 Organization

The seven remaining chapters will cover the following issues:

CHAPTER 2: Background

We review the Timethread visual notation, the formal language LOTOS, and the LARG model for architectural graphs. Several terminology definitions are also given.

CHAPTER 3: The Approach and the Contributions

We present the approach taken by the FMD research group for the integration of formal methods in the design process. This approach is based on formal interpretation methods. Four sub-methods (map decomposition, LAEG, mapping, and composition) are introduced. Then, the contributions of the thesis are enumerated w.r.t. the objectives of the previous section. Finally, we present an ongoing case study: the *Traveler* system.

CHAPTER 4: From Timethreads to LOTOS

This chapter presents the LOTOS semantics given to the Timethread notation. It enumerates a few basic concerns and according solutions, and then discusses a new timethread grammar. Single timethreads, simple interactions, and special timethread symbols are developed using this grammar, for which mapping rules are given for the generation of LOTOS specifications. Examples inspired from the *Traveler* system are given all along the chapter.

CHAPTER 5: Elements of a Life-Cycle Methodology

We firstly present a short overview of a timethread-oriented life-cycle methodology, and then different techniques related to this methodology are discussed. We present the complete mapping procedure of the *Traveler* timethread map onto LOTOS, followed by a discussion about a few transformation techniques. Finally, we apply LOTOS-based validation techniques to our timethread-oriented specification.

CHAPTER 6: Case Study: Telepresence A Multimedia System Design Example

The methods and techniques introduced in the previous chapters are applied to a more realistic real-time and distributed system: the multimedia application *Telepresence*. We present two timethread maps where one is a transformation of the other. The first one is constructed from basic use cases, mapped onto LOTOS, and then validated against the requirements. The second timethread map is a transformation (factoring) of the first one that preserves path equivalence. Again, a LOTOS specification is generated and validated using different simulation and testing techniques.

CHAPTER 7: Discussion

This chapter discusses several issues encountered in the research work of this thesis. We chose to emphasize four main issues related to: the first architecture, the STDL grammar, validation in general, and a few ideas on possible tools.

CHAPTER 8: Conclusion and Future Work

This last chapter concludes the thesis. It reviews the contributions with respect to the thesis objectives. Then, short-term and long-term research issues are identified.

CHAPTER 2 Background

2.1 The Timethread Notation

“A timethread is a *path* for the flow of causality from a *stimulus* at the start (indicated by a filled circle), through a progression of responsibilities (shown as labelled points along the path), culminating in a *response* (a “T”-junction at the end). The name (timethreads) comes from the fact that *time* increases along them and they look like *threads*” [BuC 94a]. Timethreads are useful for design discovery and system reasoning at a global and high-level perspective. They express cause-to-effect relationships by linking activities performed by the system, resulting from some stimulus (cause) and terminating with some eventual response (effect).

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. We use timethreads to focus on the end-to-end behaviour of the system we want to design. Timethreads are also *example-oriented*, meaning that they are used to show examples of representative scenarios. They are not intended to show complete behaviours, but they can clearly specify possible causality paths in the system.

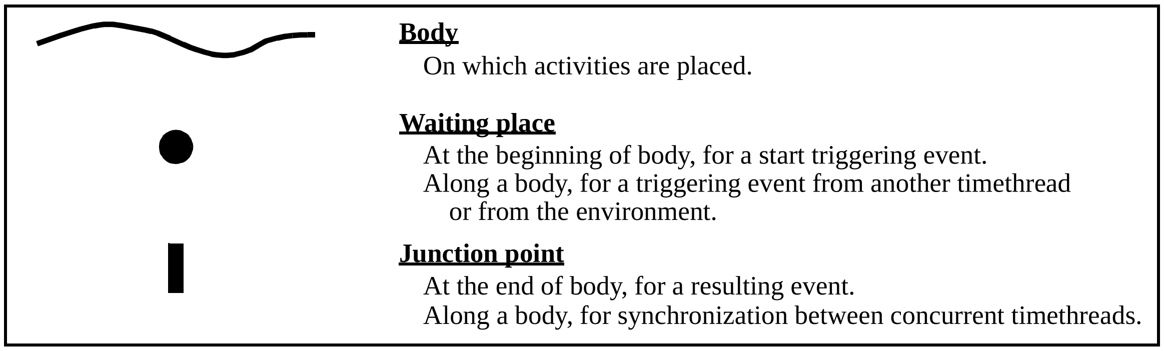
Timethreads are also different than threads of control, making them more appropriate for understanding macroscopic behaviours [BuC 92]. *Causality flow* should be interpreted neither as *data flow* nor as *control flow*. Timethreads are in fact more related to *use cases* [Jac 91] as they both represent the start and end points of causal flow paths [BCP 93].

“The flow model of timethreads is easy to understand in terms of moving *tokens*” [BuC 94b]. We can think of instances of a timethread as tokens going along its path. A token placed at the start of the path is moved along this path from responsibility to responsibility, to explain what will happen as a result of a stimulus. This token is removed when it reaches the end of the timethread. We will refer to this model as we explain how the different timethread constructors work.

Being still in evolution and not completely formal, the Timethread notation presents many open-ended issues yet it offers much flexibility to the designer.

2.1.1 Basic Timethread Set

The Timethread notation includes very few basic symbols [BuC 94a, BuC 94b, and Buh 93]. Figure 1 shows the basic notation elements of timethreads. A typical timethread starts with a *waiting place* (triggering event) and ends with a *junction point* (resulting event). The *body*, on which *activities* are placed, links the triggering event to its resulting event(s).



Body
On which activities are placed.

Waiting place
At the beginning of body, for a start triggering event.
Along a body, for a triggering event from another timethread
or from the environment.

Junction point
At the end of body, for a resulting event.
Along a body, for synchronization between concurrent timethreads.

Figure 1: Basic notation elements

More complex timethreads are also composed of these symbols. Notions such as choice, parallelism, and synchronization can also easily be expressed. Figure 2 describes the usual ways of forking and joining timethread paths. In this figure, we assume a timethread flow from left to right:

(*a*) OR-Fork:	An exclusive choice between two paths is given. A token along the entering path will follow only one of the exiting paths.
(*b*) OR-Join:	Two paths merge into one, without any synchronization. A token along any of the entering paths will follow the single exiting path.
(*c*) AND-Fork:	A path forks into two concurrent paths. A single token along the entering path will split into clones that follow each exiting path concurrently.
(*d*) AND-Join:	Two paths synchronize together and only one path results. One token from each entering path will wait one from each other path. They combine afterwards in a single token to follow the exiting path.

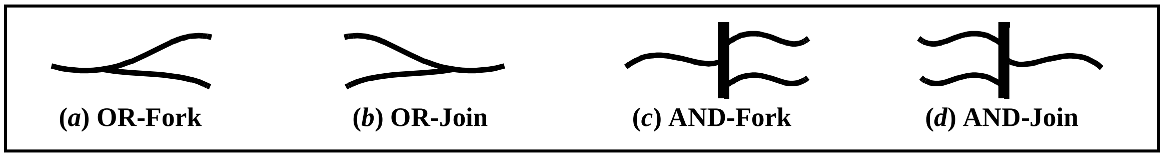


Figure 2: Forks and joins

Note here that there is no constraint on the number of paths involved. We may join more than two paths at once, and they could come from the same timethread or different ones. The same reasoning applies to the fork operations.

2.1.2 Timethreads Interactions

Synchronous and asynchronous interactions, implying two or more timethreads, are easily expressed without adding any new notation. Interactions may occur on waiting places and junction points. The most common types of interaction are presented in figure 3.

(*a*) Concatenation:	An ending timethread triggers another one. A token at the end of its path is removed and a new one is placed at the beginning of the second timethread and follows it.

(*b*) In-passing: Asynchronous interaction where the timethread on the left triggers the one on the right, without stopping, and then continues. A new token is placed at the beginning of the timethread on the right when the token from the timethread on the left passes the start point. After that, the two tokens progress concurrently down the two separate paths.

(*c*) OR-Start: A timethread is triggered by one or the other ending timethread. A token at the end of either ending path is removed and a new one is immediately placed at the beginning of the starting timethread and follows it.

(*d*) AND-Start: Two timethreads synchronize together and trigger another one. One token from each ending path will wait for all others an then they allow a new token to go on the next timethread path.

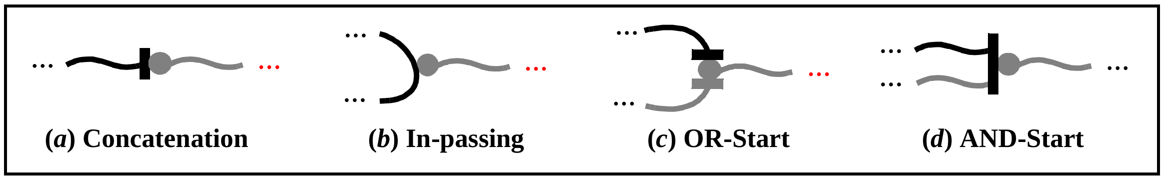


Figure 3: Typical interactions

Note that the number of interacting timethreads is not restricted to two or three. In the general case, we can compose these types of interactions to build more complex ones involving many more timethreads. Also, interactions can occur at any waiting place along a timethread, not only at the starting event (see figures 40 and 41). This approach is flexible and yet powerful.

2.1.3 Other Symbols

Special-purpose symbols can be added at the designer's convenience to increase the expressiveness of the Timethread notation. In the literature [BuC 94b], many such symbols have been introduced, especially to highlight issues associated with robustness, real-time, and concurrent behaviour that need to be resolved in the architecture. Because of their usefulness in many situations, some of these symbols will receive special attention in chapter 4. They are presented here in figure 4:

(*a*) Timer: A special waiting-place that is used to express delay, time-outs, watch-dogs, etc.

(*b*) Stub:	A collapsed timethread that is to be defined (or refined) at a later stage. Details are hidden intentionally at this level of abstraction. Stubs may also have other uses.
(*c*) Abort:	Destroys the instances of another timethread (or the tokens along the timethread).
(*d*) Loss:	This ‘ground’ symbol indicates the loss of a token along the timethread path. It is used to express robustness concerns.

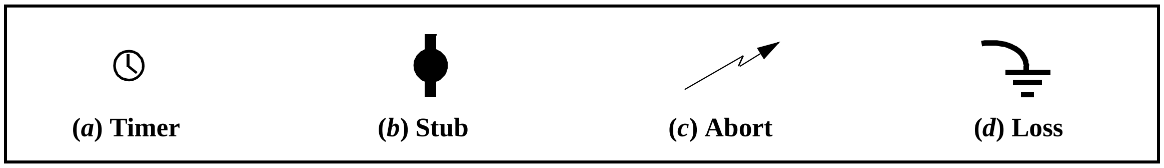


Figure 4: Other symbols

2.1.4 Timethreads Refinement and Transformations

Different timethreads refinement and transformations have been used in the literature. They help the designer to manipulate a timethread map. Although they are not completly formalized yet, the most important ones are presented here in a general way (see [BoL 94] for further information):

- *Activity refinement*: An activity is considered as a *black box* which can itself be decomposed as a sequence of activities.
- *Stub refinement*: More general case where a black box (timethread stub or path stub) is replaced by a more complex and complete timethread or path.
- *Functionality extension*: Addition of details to a path (concurrent or alternative path, new activities, etc).
- *Timethread cutting*: A timethread is split into two or more independent timethreads.
- *Timethread merging*: Independent timethreads are merged together to form only one timethread. Parts of paths from different timethreads can also be merged.
- *Timethreads composition*: Addition of an interaction between several timethreads.
- *Timethreads decomposition*: Removal of an interaction between several timethreads.

2.2 LOTOS

In this section, we recall the origin of LOTOS as a formal technique, and then present its main operators, transformation and validation concepts, and a few tools used in our research. For a more complete description of the language and its different uses, refer to [BoB 87, LFH 91, Sch 93, and Tur 92].

2.2.1 Formal Methods

Formal methods, in particular process algebras, proved their usefulness in capturing descriptions of complex, concurrent, and communicating systems. LOTOS (Language Of Temporal Ordering Specification) is an algebraic specification language and a Formal Description Technique (FDT). It was especially developed for the formal description of the OSI architecture (interfaces, services and protocols), although it is applicable to distributed and concurrent systems in general. Today, people try to extend its field of action on hardware, telephony [Bou 91], operating systems, embedded systems and real-time systems. LOTOS has been an ISO Standard (8807) since 1989 [ISO 88].

The basic idea of LOTOS is to describe a system by defining the temporal relations along the interactions that constitutes the system’s externally observable behaviour. The process part of LOTOS (known as *Basic LOTOS*) is based on ideas found in CCS [Mil 80] and CSP [Hoa 85]. The data part of LOTOS (included in *Full LOTOS*) is based on the theory of abstract data types and comes from the language ACT ONE [EhM 85].

2.2.2 Operators

In LOTOS, systems are described in terms of *processes*. A process is viewed as a black box interacting with its *environment* via its observable *gates* (figure 5). Its internal actions are unobservable by the environment. The *behaviour expression* is built by combining LOTOS actions by means of operators and possibly instantiations of other processes.

The basic element of a behaviour expression is the *action* which represents a synchronization between processes, between a process and its environment, or both. An action consists of a gate name, a list (possibly empty) of value experiment offers (value offers or interaction parameters), and possibly a predicate that imposes conditions on the event to be accepted. Actions are *atomic* in a sense that they occur instantaneously, without consuming time.

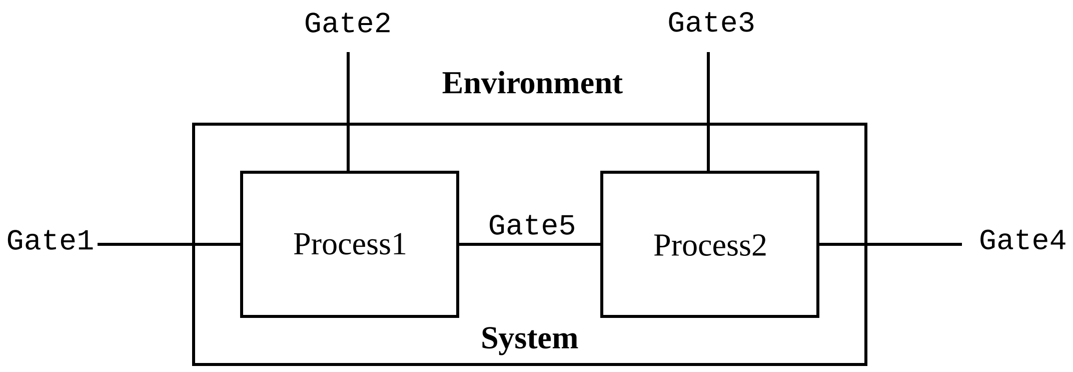


Figure 5: Representation of a system specified in LOTOS

In figure 5, the system is composed of two processes that interact with each other on the hidden gate `Gate5` (interaction point). In LOTOS terms, we say that `Process1` is synchronized with `Process2` on `Gate5`. LOTOS synchronization is based on a multi-way rendezvous concept.

The partial LOTOS specification 1, corresponding to the system presented in figure 5, is given here (reserved words are in **bold**):

```
specification System [Gate1, Gate2, Gate3, Gate4] : noexit
behaviour
     hide Gate5 in
          Process1[Gate1, Gate2, Gate5]
          |[Gate5]|
          Process2[Gate3, Gate4, Gate5]
where
     process Process1[Gate1, Gate2, Gate5] :noexit :=
          (* ... Behaviour of Process1 *)
     endproc
     process Process2[Gate3, Gate4, Gate5] :noexit :=
          (* ... Behaviour of Process2 *)
     endproc
endspec
```

Specification 1: System example

The main LOTOS constructors are recalled in figure 6, where `a` is an action, B_i are behaviour expressions, g_i are gates, and `P` is a predicate:

	Name	Behaviour Expression	Comment
Basic Behaviour Expressions	Inaction	**`stop`**	Cannot engage in any interaction (deadlock).
	Successful Termination	**`exit`**	Indicates that a process has sucessfully performed all its actions.
	Process Instantiation	`ProcName [`g_1`, ..., `g_n`]`	Creates an instance of a process.
Basic Operators	Action Prefix	`a; B`	Used to prefix a behaviour expression `B` with an action `a`. There exists a special action, called **`i`**, that a process can execute independently.
	Choice	B_1 **`[]`** B_2	Allows the user to define different alternatives for a given process.
Enabling and Disabling	Enabling	B_1 **`>>`** B_2	Used to sequence two behaviour expressions. B_1 has to **`exit`** for B_2 to be executed.
	Disabling	B_1 **`[>`** B_2	Used to express situations where B_1 can be interrupted by B_2 during normal functionning.
Composition	Parallel Composition	B_1 **`|[`**`g1, ..., gn`**`]|`** B_2	Composition in which B_1 and B_2 behave independently, except for the gates g_1, ..., g_n where B_1 and B_2 must synchronize.
	Interleaving	B_1 **`|||`** B_2	Composition in which B_1 and B_2 behave independently (the synchronization set is empty).
	Full Synchronization	B_1 **`||`** B_2	Composition in which B_1 and B_2 are synchronized on all their gates.
Other Operators	Hiding	**`hide`** g_1`, ..., `g_n **`in`** `B`	Used to hide actions (g_1, ..., g_n) which are internal to a system. These actions cannot synchronize with the environment.
	Guarded Behaviour	**`[`**`P`**`] ->`** `B`	`B` can be executed if `P` is true.
	Local Definition	**`let`** `x:s = E` **`in`** `B`	Substitutes a value expression (`E`) by a variable identifier (`x`) of sort `s` in `B`.

Figure 6: Main LOTOS constructors

More operators exist (`choice` and `par`) but they are not used in this thesis. Also, the construction and the use of abstract data types in LOTOS are not discussed in the current section, but they are fully described in [ISO 88].

2.2.3 Transformations and Validation

LOTOS provides formal means to manipulate, combine, factor, and transform behaviour expressions in various ways. Properties can also be verified and tested by the available tools. We present here the terminology used in the LOTOS community.

Transformations

"The term *transformation* is often used to cover all forms of refinement and reformulation of a specification" [Tur 91]. For instance, a high-level abstract specification can be transformed into a more concrete and deterministic one. This is usually referred as *stepwise refinement*. LOTOS allows many types of transformations from which *Correctness Preserving Transformations* [CPT 92], or *CPTs*, are the most interesting ones. There exist many CPTs such as functionality decomposition, behaviour expansion, action refinement, processes splitting and regrouping, gates rearrangement, inverse expansion, multi-way to two-way synchronization, etc. Transformations from a specification written in one LOTOS style to a specification in another style also exist [VSS 91]. Equivalence, reduction and extension relations are used to assess the correctness of the transformations.

Validation

Demonstration of design compliance with stated and unstated user requirements [LOT 92]. *Validation* is a generic term that includes testing and verification techniques. It is mostly used to check properties such as conformance, absence of deadlocks, liveness, and completeness.

Testing

Checking of the real behaviour of the system by the application of test cases [LOT 92]. Practical *testing* (which is not exhaustive), is confined to the detection of certain types of problems or particular instances of inconsistencies. A testing theory for LOTOS is presented in [Bri 88]. It includes notions of canonical testers, conformance testing and test cases derivation.

Simulation

LOTOS specifications are executable. *Interactive simulation* is therefore a validation technique often used, mainly in the early stages of the design process.

Verification

“Demonstration of consistency between two designs” [LOT 92]. The algebraic properties of LOTOS provide a theoretical foundation that supports some formal reasoning about the specification, called *verification*. Behavioral specifications can be verified using relations such as bisimulation, testing equivalence, and trace equivalence. Desirable properties for the system, expressed in terms of temporal logic formulas, can be verified using model checking [Grh 92].

These definitions of validation and verification slightly differ from the ones now generally used by the software engineering community. In [Pre 87] for instance, verification is defined as “Are we building the product right?”, and validation as “Are we building the right product?”. The terminology used in the thesis is based on the definitions given in the previous paragraphs.

2.2.4 Tools

LOTOS tools for various stages of the development cycle are developed by many groups around the world [GLO 91 and Sch 93]. From the tools available to us, two are particularly useful for step-by-step execution of specifications:

ELUDO

ELUDO (Environnement LOTOS de l’Université D’Ottawa) is a toolkit regrouping many previous tools (such as ISLA and SELA) with a new interface. An X-Windows interface also exists (XELUDO). We will use ELUDO for validation purposes in the upcoming case studies.

LOLA

LOLA (LOTOS LAboratory) is a tool developed at the Universidad Politécnica de Madrid. It also allows the simulation and expansion of LOTOS specifications.

Other tools, such as *SMILE*, *CAESAR*, and *TOPO*, also possess the functionalities required to ‘animate’ the specifications, just as it is needed in the case studies.

2.3 LARG

2.3.1 The LARG Model

The *LARG* (LOTOS Architectural Representation Graph) model has been developed to serve as the intermediate structural model in the LOTOS interpretation method for architecture-based design [Bor 93 and BBO 94]. An example of a LARG, in which the different types of components are identified, is given in figure 7. The structural components of the LARG model are called *process*es. Interactions between processes are realized by means of multi-way rendezvous on *gates*.

The initial LARG model has been developed in such a way that the LARG artifacts (processes and gates) can directly be mapped onto LOTOS structural constructs. Finally, for the purpose of the *LAEG* (LOTOS Architectural Expression Generation) method, both a Grouping algorithm and an UnGrouping algorithm have been defined on LARGs. The LARG model, the Grouping algorithm and the UnGrouping algorithm are all formally defined in [Bor 93].

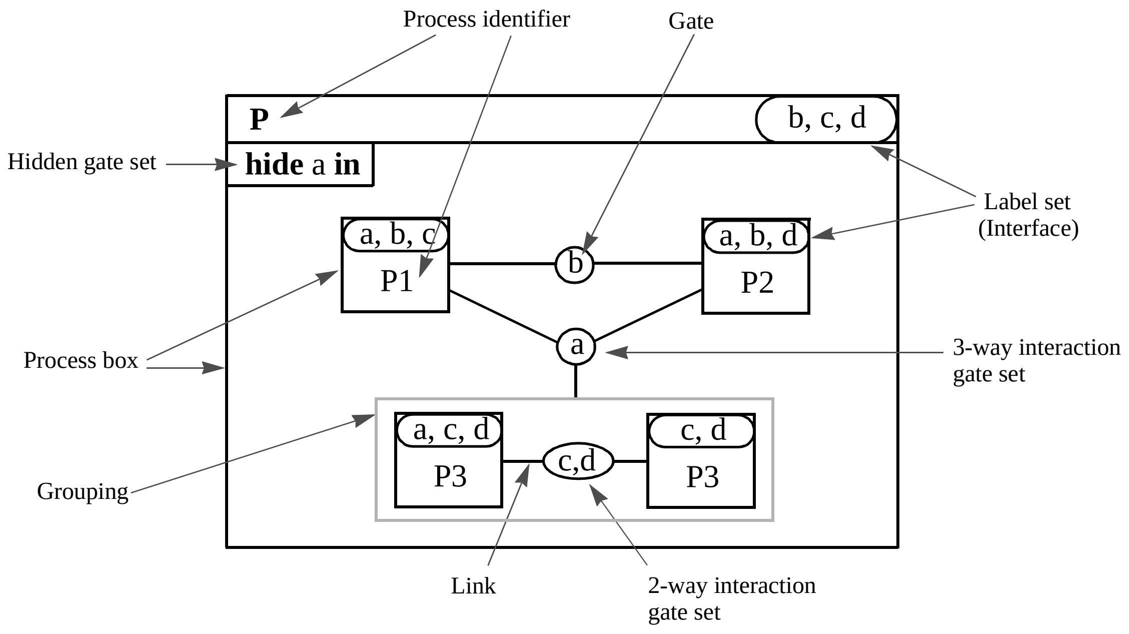


Figure 7: Example of a LARG

2.3.2 LAEG Method

The LAEG method aims at generating LOTOS structural expressions from LARGs. It is conducted in two distinct phases:

- LARG analysis
- Generation of LOTOS structural expressions.

LARG Analysis

This phase aims at detecting architectural errors and non-determinism. In the case of timethreads interpretation, the LARG analysis phase can be reduced to a structural ambiguity identification (called *non-deterministic interaction-choice* in [BoA 93]).

We say that a gate *g* is the source of structural ambiguity in a LARG *P*, iff:

- *g* is contained in more than one gate set (GS) in *P*
- Every GS containing *g* is linked on one side to a constant set of processes, called the *root process set* of the structural ambiguity, and on the other side to distinct processes, i.e. processes which are linked to only one GS containing *g*, called the *choice process set* of the structural ambiguity.

Thus, every process which possesses gate *g* is either linked to every GS containing *g*, or to one and only one GS containing *g*.

In figure 8, an example of such structural ambiguity is given. In this LARG, gate *a* is the ambiguous gate. We observe that *P1* can interact with either *P2* or *P3* on the 2-way interaction gate *a*. We also observe that *P2* and *P3* do not interact together. Therefore, in order to have an interaction on gate *a*, we need to have *P1* ready to interact on *a* and either *P2* or *P3* also ready to interact on *a*.

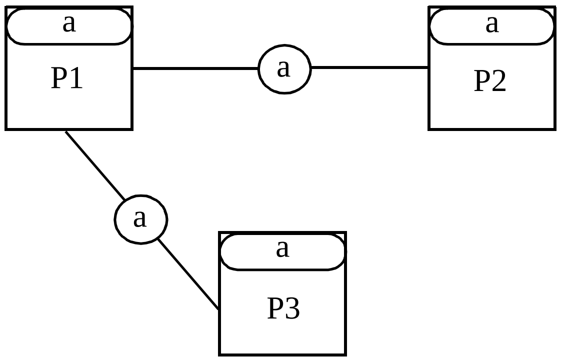


Figure 8: Structural ambiguity in a LARG

Generation of LOTOS Structural Expressions

The second phase consists in generating LOTOS structural expressions from LARGs. This phase is essential since LOTOS possesses binary operators only. It involves successive applications of the Grouping algorithm. The algorithm is applied until we obtain a binary grouped LARG which is equivalent to the former one.

An illustration of LARG binary grouping is given in figure 9. Figure 9(b) gives an equivalent binary grouping LARG which has been obtained by successive applications of the Grouping algorithm. The grouping sequence used in figure 9 has been arbitrarily chosen, and is only one of many possible solutions.

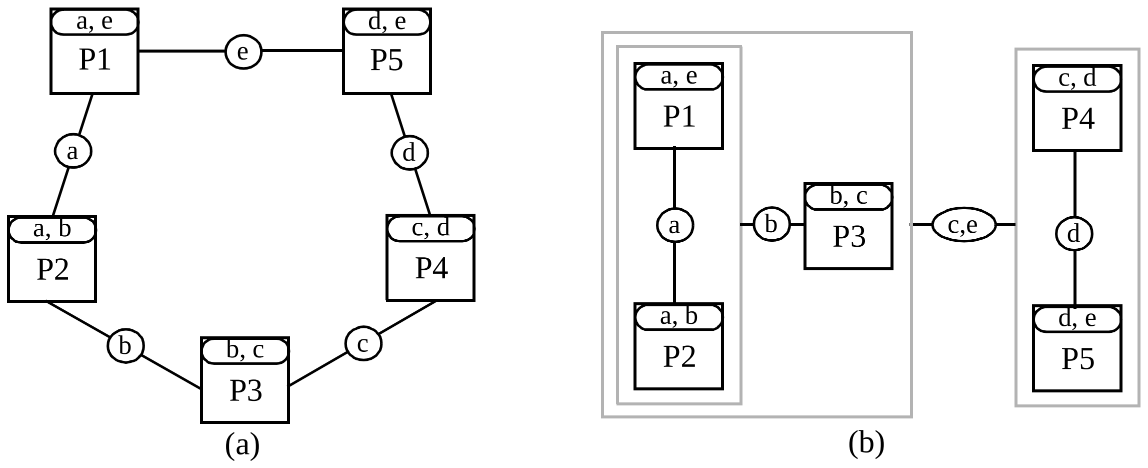
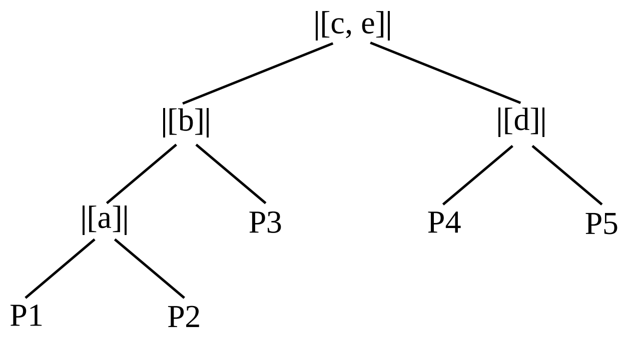


Figure 9: Binary grouping of a LARG

The tree representation of the LARG of figure 9(b) and its associated LOTOS structural expression are given in figure 10. We see from these two figures that the generation of a LOTOS structural expression from a binary grouped LARG is straightforward.:



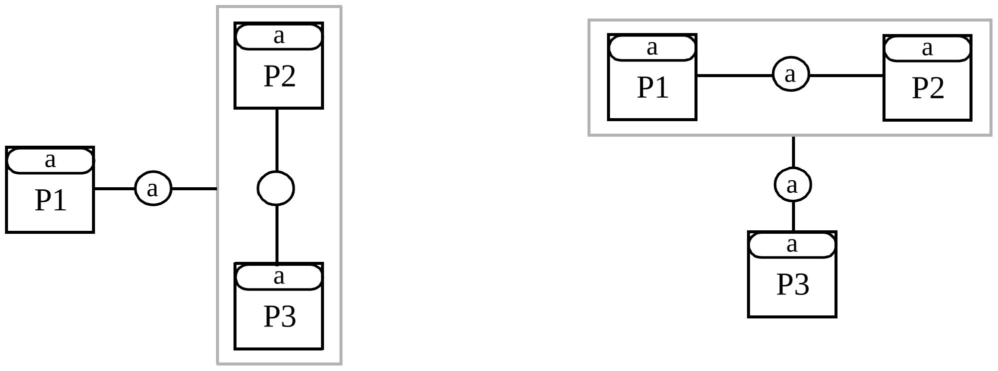
((P1[a, e] |[a]| P2[a, b]) |[b]| P3[b, c]) |[c, e]| (P4[c, d] |[d]| P5[d, e])

Figure 10: Tree representation and LOTOS A.E. of the linearized LARG

Structural Ambiguities in LARG

Grouping an ambiguous LARG might be problematic because, in some cases, groupings violate the interaction semantics of the LARG. For example figure 11(a) and 11(b) represent two different groupings derived from figure 8. We observe that these two groupings lead to two non-equivalent LARGs. In the first case *a* is a 2-way interaction while in the second case *a* is a 3-way interaction. The LARG of figure 11(a) corresponds to a correct interpretation of figure 8, while figure 11(b) corresponds to an incorrect one.

To eliminate ambiguities from such LARGs, a technique called *structural ambiguity grouping* (also called *non-deterministic interaction-choice grouping* in [Bor 93]) was defined. In this technique, we group together all processes contained in the choice process set (refer to [Bor 93] for more details on grouping techniques). Figure 11(a) illustrates an example of the application of this technique.



(a) Structural ambiguity grouping

P1[a] |[a]| (P2[a] ||| P3[a])

(b) Incorrect grouping

P3[a] |[a]| (P1[a] |[a]| P2[a])

Figure 11: Grouping an ambiguous LARG

2.4 Definitions

Since LOTOS and the Timethread notation use common terminology, we define here the specific terminology that will be used in the remaining chapters:

Triggering event:	Starting event of a timethread.
Resulting event:	Ending event, termination of a timethread.
Process:	A LOTOS behaviour abstraction, unless cited otherwise.
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.

In the thesis, new words or concepts, as well as references to timethreads and LOTOS code, will be *italicized.*

CHAPTER 3 The Approach and the Contributions

Five requirements (referred as *R1* to *R5*) have been defined in section 1.1. Many problems might arise from such general requirements. The following sections present a solution based on the Timethread notation, the formal description technique LOTOS, and formal interpretation methods. Then, the contributions are introduced with respect to the thesis objectives.

3.1 The Approach

A solution to our problem is introduced in [Bor 93 and BBO 94], where the concept of *formal interpretation methods* is presented. Such a method allows the interpretation of a given design in terms of a given formal semantic model. We think we can apply this idea to the Timethread notation.

Timethreads are an intuitive visual notation that can be used as a design model to capture the requirements (*R4* in §1.1). They can also lead to a first architecture expressed in any design description model useful to designers (*R1*). This introduction of timethreads in the design process (figure 12) is simple and yet very helpful for designers since they already use most of these concepts, often in an ad-hoc way. Such integration facilitates the transition from the problem domain to the solution domain.

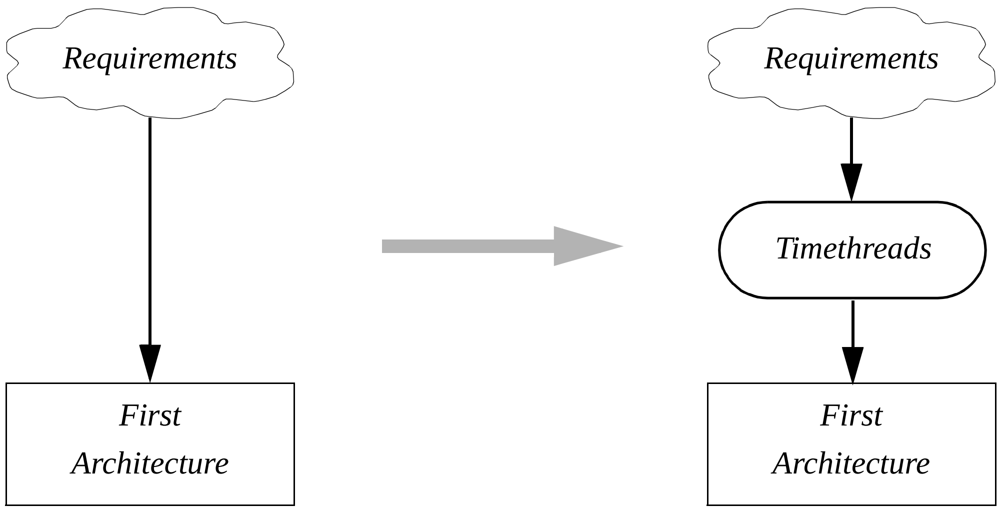


Figure 12: Timethreads in the design process

Being still informal, timethreads open the door to the introduction of formal methods in the design process. We can create an *interpretation method* for timethreads, allowing their interpretation in one or more formal languages (*R2*).

LOTOS is the first formal method that has been chosen as a formal basis for our Timethread notation. Previous work has been done on other approaches. In [ViB 91] and [Vig 92], 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* at that time) and LOTOS. In [LaB 92], the authors try to see whether or not two different approaches of a design conception, ObjecTime and LOTOS, could be used in a complementary way in order to add timethreads concepts to the ObjecTime tool. The approach presented in our research differs considerably from these two, but the experience gained helped in getting a better understanding of timethreads.

Other formal methods, such as Petri nets [BDC 92] and event structures [Roz 92], can be considered as other options as a formal basis for the Timethread notation. In [FCB 93], the authors introduced a Petri net interpretation of timethreads. As a first step, we decided to use LOTOS. Its expressiveness (especially w.r.t. communication), its transformation and validation methods, and the numerous available tools are among the main reasons why we chose this particular language.

In order to integrate LOTOS in the timethread-centered design process, the first step consisted in defining an interpretation method that allows the generation of LOTOS specifications from *timethread maps*, which are collections of interacting timethreads. The idea of formal interpretation method presented in [Bor 93] has been adapted to timethreads and LOTOS by F. Bordeleau in [BoA 93]. We present this method in figure 13 where the grey box represents the main contributions of this thesis.

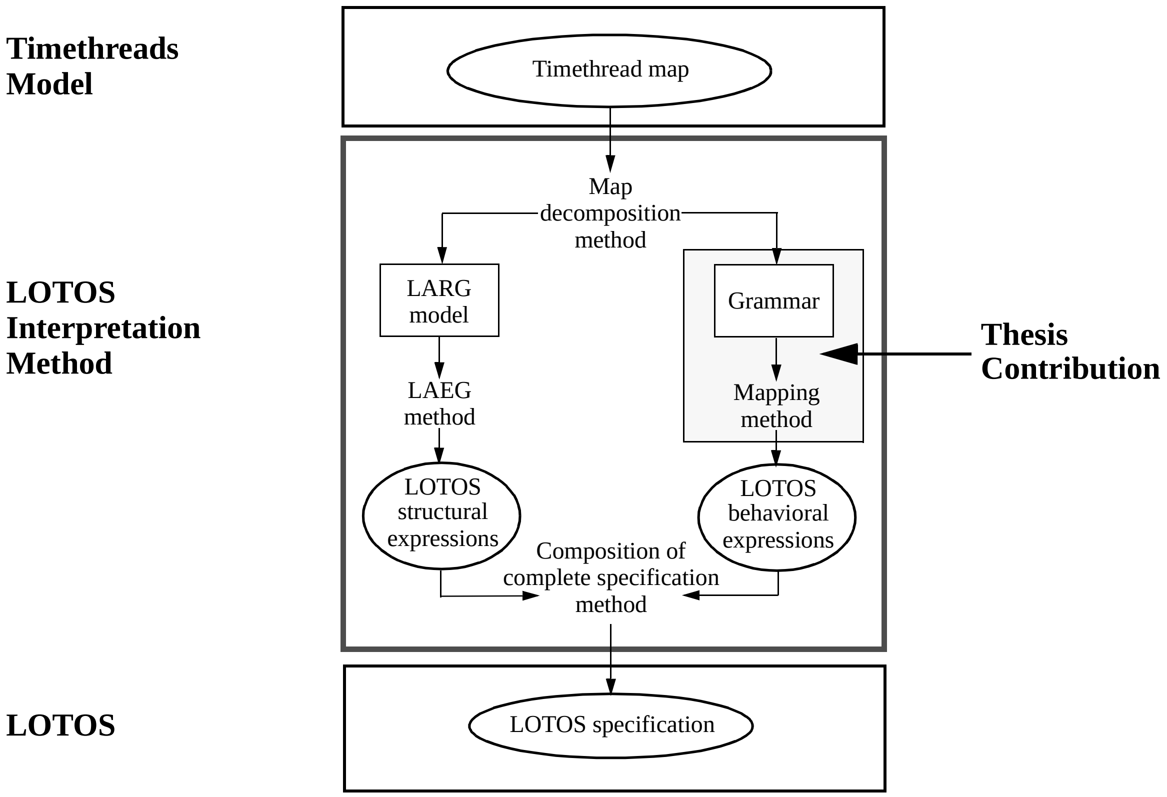


Figure 13: LOTOS interpretation method for timethreads

This *LOTOS interpretation method for timethreads* is composed of four methods, which are enumerated here:

Map decomposition method

In our view, timethreads are considered as entities in their own right. This leads to a decomposition method consisting of two steps:

- Mapping of the topology of interacting timethreads (from the map) onto a LARG. This is mostly discussed in section 4.4.
- Description of the paths of individual timethreads using a timethread *grammar.*

The LARG and the grammar are considered part of the internal representation of the timethread map. Therefore, this thesis will take the LARG and the grammar as starting points for the generation of a specification. The grammar will be discussed in chapters 4 and 7.

LAEG Method

The LAEG (LOTOS Architectural Expression Generation) method aims at generating LOTOS structural expressions from LARGs. It is conducted in two distinct phases:

- LARG analysis, where potential structural ambiguities are detected and fixed.
- Generation of LOTOS structural expressions.

This method was presented in §2.3.2.

Mapping Method

The mapping method is a compilation from the grammar representation to a LOTOS behaviour expression. In the next chapter, this compilation process is shortly introduced, and many examples and rules are developed, but no complete algorithm will be given. This is still work to be done.

Composition of the Complete Specification Method

The composition of the complete specification method consists in combining both the LOTOS structural expression (which expresses the way timethreads interact in the timethread map) and the different LOTOS behavioral expressions (each of which expresses the activity sequence in a single timethread) in a global LOTOS specification. The resulting LOTOS specification reflects the path behaviour of the complete timethread map and can be used as an input to validation tools.

3.2 Contributions of the Thesis

The major contributions of this thesis related to the thesis objectives (*O1* to *O4* in §1.2) are enumerated here.

LOTOS Interpretation of Timethreads

A LOTOS interpretation method for *individual* timethreads is developed. It allows the generation of LOTOS processes from single timethread. This interpretation provides a formal semantics to the Timethread notation, with LOTOS as an underlying model. This contribution intends to satisfy the objective *O1* which relates to the generation of LOTOS specifications from timethread maps.

Timethread Grammar

A grammar for single timethreads description is presented. This context-free grammar is an internal representation that defines the single timethreads in a map, and allows the generation of specifications in formal languages such as LOTOS. Objectives *O1* and *O3* (support of tools) are aimed by the creation of this grammar.

Techniques

Techniques for the transformation of timethread maps are introduced, and validation techniques are discussed. They are mostly based on LOTOS transformations and validation techniques, and on different LOTOS tools. These techniques should help designers to play the design (objective *O2*).

Case Study

A multimedia case study, the *Telepresence* system, is developed in chapter 6. The interpretation method is applied to the timethread map in order to get a LOTOS specifications. We also make use of the transformation and validation techniques on this example. This complex case study also attempts to satisfy all four objectives.

Of course, for each contribution, we will point out resulting problems, difficulties, and research issues (objective *O4*). Other minor contributions will be identified along the remaining chapters.

We believe that the approach we propose will help in capturing and testing system requirements. Also, once user-friendly timethread interfaces are available, our method could lead to fast production of formal specifications in industry, thus allowing designers to use the power of formal techniques.

3.3 Ongoing Case Study for Chapters 4 and 5

An ongoing case study is used in chapters 4 and 5 in order to relate methods and concepts to a concrete and simple example. For this purpose, we use the *Traveler system*, an example adapted from [BuC 93] and [BuC 94b]. A complete LOTOS specification will be derived from the timethread map of the traveler system using the LOTOS interpretation method for timethreads described in this thesis (see also [BoA 93]).

The traveler system, shown in figure 15, is not a computer system in a literal sense. This example depicts a familiar situation from everyday life which is easy enough to illustrate properties similar to common computer systems. We can think of the travelers, the taxis, the planes, etc., as components analog to computer-based subsystems, processes, or objects. Therefore, the traveler system will help us thinking about distributed systems in the large without committing to any architectural concerns.

Note that this is not *the* only way of using timethreads for design. For instance, timethreads have been used in association with role-architectures in object-oriented design [BuC 94b].

3.3.1 Informal Description of the Traveler System

Travelers use a *traveler system* to get to a certain destination. The timethread map of figure 14 shows a *use case* [JaA 92] delimiting the system (black box) and its environment. We consider this a use case because it expresses a sequence of transactions in a dialogue between the user and the system.

To transform this black box into a grey box showing how a traveler gets to its destination, we need a more complete description. The latter will be our starting point for the development of the interpretation method.

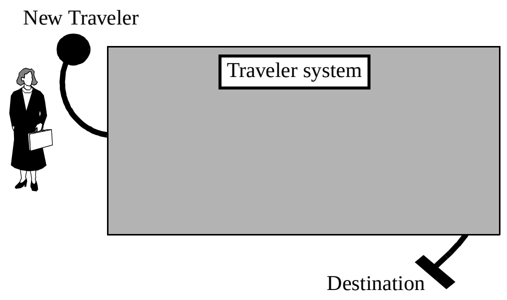


Figure 14: Use case of the traveler system

Suppose that the traveler system is composed of a taxi company, where a dispatcher receives requirements from the travelers and then dispatches a taxi, and an airline. Different components are defined: traveler, dispatcher, cab, and plane. They collaborate to get travelers to their destination without the intervention of a master controller to direct their individual activities and without themselves necessarily having individual knowledge of how they fit into the whole [BuC 93]. This can be considered a distributed system.

Here is the path description of each component (fig. 15), with corresponding activities along the path. When a new traveler comes (*Tnew*), she phones the dispatcher for a cab (*TphoneD*), goes to a rendezvous point, gets in the cab (*TgetinC*), has a taxi ride (*TCride*), gets out the cab (*TgetoutC*), and goes to the airport (*Tairport*). Then, she waits for a plane, gets on the plane (*TgetonP*), flights to another airport (*TPflight*), gets off the plane (*TgetoffP*), and finally gets to the final destination (*Tdest*).

The dispatcher comes to the office (*Din*), waits for a request from a traveler (*TphoneD*), looks for an available cab (*DlookforC*), asks for a cab (*DaskC*), fills internal statistics (*Dfillstats*), and leaves the office (*Din*) or gets ready for the next traveler (*Dready*).

A taxi driver gets in the cab (*Cin*), waits for a request from the dispatcher (*DaskC*), waits for the traveler to get in at a rendezvous point (*TgetinC*), gives a ride to the traveler (*TCride*), leaves the traveler (and gets paid!) (*TgetoutC*), and gets ready for a new request (*CgoD*) or goes to the garage (*Cgarage*) and gets out the taxi (*Cout*).

At the airport, when an airline plane is ready (*Pready*), it waits for a traveler to get on (*TgetonP*), flies to the next airport (*TPflight*), leaves the traveler (*TgetoffP*) and goes to a hangar (*Phangar*).

3.3.2 Timethread Map of the Traveler System

Following the complete description of the last section, the simple use case presented in figure 14 can be refined, using a timethread-centered design process [BuC 93], into a detailed path description: the timethread map of figure 15.

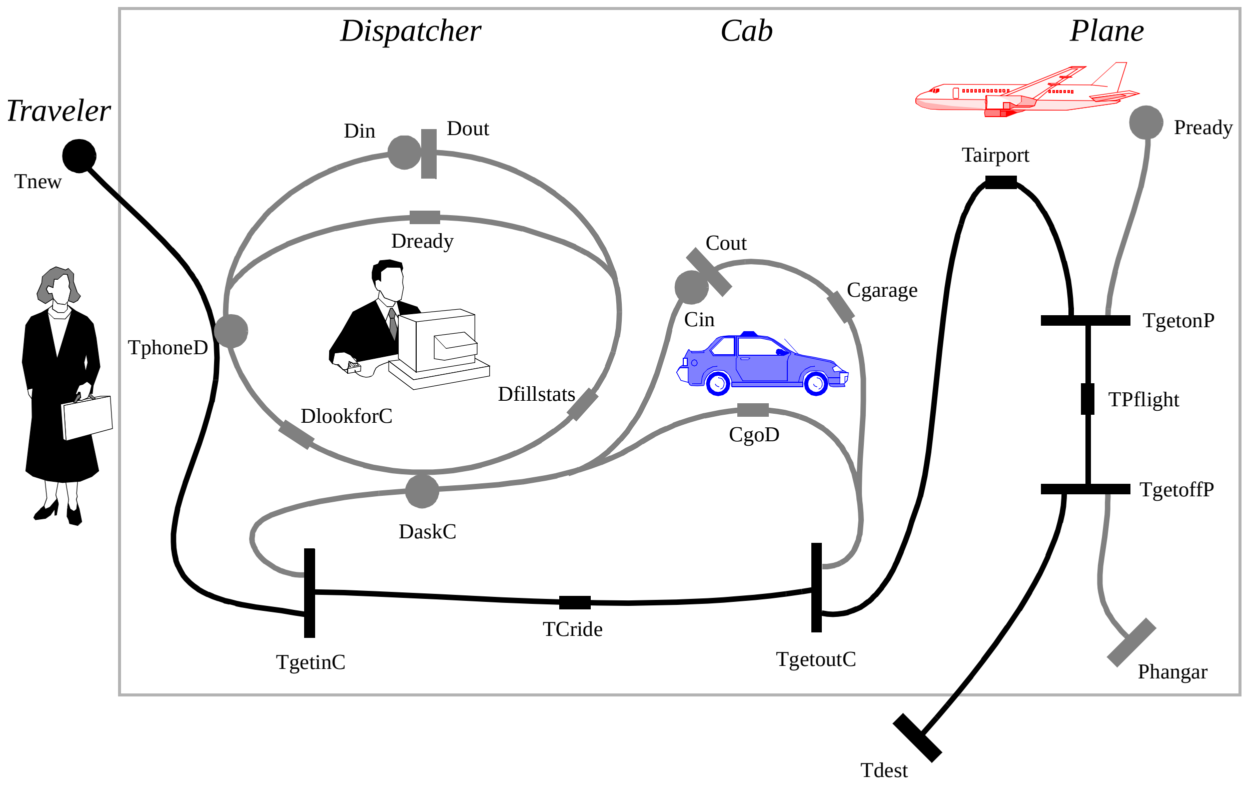


Figure 15: Timethread map of the traveler system

The refinement process is not presented here. This diagram is considered as a first “design” and a LOTOS specification can therefore be derived. A few things have to be noted here:

- The refined grey box description of the system under design (SUD) still has the same environment as the black box description (fig 14). Every activities in the SUD could be “hidden” from a LOTOS point of view.
- A timethread is neither a component, an agent, nor an object, as the map could suggest. Timethreads *span* components, and they are not necessarily related on a 1-to-1 basis with components. Therefore, the fact that we have four timethreads here and that we assumed we have four components is a coincidence.
- Different shadings are used here to differentiate timethreads, to give them a different identity. The identity of a timethread’s segment is not yet clarified in the notation. Shadings, colours, and identifiers can be used for this purpose.

CHAPTER 4 From Timethreads to LOTOS

The Timethread notation includes a basic set of timethreads symbols, symbols to denote different types of interactions between timethreads, and special symbols. This chapter presents the LOTOS semantics given to this notation.

4.1 Basic Concerns

4.1.1 Guiding Rules

We need a few guiding rules to help us giving a semantics to timethreads:

- We consider timethreads as entities in their own right. One way to represent this fact is to associate one LOTOS process to one timethread. This solution is preferable to the one where each section of a timethread path is mapped onto a LOTOS process. The latter solution leads to many processes and hidden interactions that destroy the timethreads structure.
- Waiting places and junction points are represented as LOTOS gates on which interaction with the environment or with other timethreads will occur. They can be hidden to represent abstraction levels or internal interactions.

- The body will only represent sequencing of activities. No special semantics is given to empty paths, although some underlying machinery may be refined from these paths at a later stage. Because we cannot specify what is not explicitly drawn on a timethread map, we assume no behaviour.
- Timethreads are not a front-end for LOTOS-based design. LOTOS is the formal underlying model that supports timethread design. Therefore, we do not intend to generate LOTOS in any traditional specification style [VSS 91]. A *timethread-oriented style*, which reflects the timethread structure of the system under design but not its final architecture, will result from the mapping.

With these ideas in mind, we can now proceed in giving a semantics to the Timethread notation.

4.1.2 Levels of Specifications

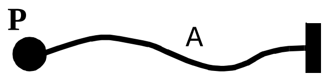


Figure 16: Basic timethread

Figure 16 represents a *basic timethread*, or a cause-to-effect relationship. It is intuitive 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 §2.4 for the terminology).

We should also consider the start point and the end point as LOTOS gates. The start point has a triggering event, called *Trigger*, coming from the environment or from another timethread. The end point has a resulting event called here *Result*. Thus, a unique instance of this timethread could be represented as follows (we deliberately omit 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 reactive systems, our timethread's representation must be able to react to more than one initial stimulus from its environment. We would like this process to be executed as often as the environment desires to, i.e., more than one token can go along the path. Hence, recursion can 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 (or the tokens to go concurrently), which is not the case in the last definition. LOTOS parallelism needs to be introduced and unbounded recursion should be avoided, as in the following process:

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

A could be an empty sequence of activities. In this case, the timethread will simply represent 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
```

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 type of recursion makes validation and execution of LOTOS specifications more difficult.

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

```
process P[...] (n:Nat): noexit :=
(* n > 0 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))* instantiates *n* instances of process *P*, as in a countdown, namely from *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* occurrences of process *P* in parallel, without any tail recursion. Therefore, only *n* concurrent instances will exist and terminate:

```
process P[...] (n:Nat): noexit :=
(* n > 0 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 is a parametrization where we have a bounded number (*n*) of instances, executed sequentially:

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

Thus, several different types of behaviours can be associated with a timethread. Depending on what exact behaviour we want to simulate, different levels of abstractions can be defined.

Figure 17 presents a summary of *options* associated to our levels of specification. A short example (without gate parameters) is given for each:

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

Level	**Options**	**Example**

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

Figure 17: Levels of abstraction and their options.

Depending on what type of questions we want to ask of 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 in early stages of the design process, a level 1 (*L1*) specification is rapidly generated and tested. For more complex and realistic scenarios (including concurrency, robustness, cycles, etc...) or for the generation of test cases for the implementation, a level 2 specification could be used. The last level (*L3*) is like a level 2 specification where there is no commitment to a specific number of instances. Note that (*L3*) has semantics nearly equivalent to the Petri nets presented in [FCB 93], while (*L1*) leads to more workable and understandable 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*.

Our interpretation of timethreads often results in a new style of LOTOS code, i.e., with a lot of concurrent instances and many resulting **`stop`** processes. This *timethread-oriented style* reflects the timethread structure of the system under design but not its final architecture. We are concerned here with a behavioral interpretation of the path specification, without architectural considerations (at least at this abstraction level).

By 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 (such as 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 meta-language (or a tool control language) to manage the number of instances of a process. No option would be needed with such an operator: a single level of specification could always be used for any simulation. This feature is not implemented in any known tool yet, and therefore we have to simulate it directly in the specification.

In this thesis, we mostly use level 1 specifications, because they are the most simple and useful ones. Level 3 specifications are discussed sometimes, but level 2 specifications are put aside because they introduce a high level of complexity in LOTOS specifications for a very little gain.

4.1.3 Tag Mechanism

In the previous section, we mentioned that each timethread instance, or token, has an implicit state. This state could determine which path will be followed when a choice occurs. We may or may not know this state during early stages of the design process. But at some point in time, we want to capture it to give a more specific picture of behaviour.

Timethreads are path specifications, not complete behaviour specifications. Although the intention is not to specify the complete behaviour, if we can be accurate on what path can be taken, based on an instance’s internal state, then we should use this information in the diagram to derive the specification accordingly. The notation may need to have *tags* and *guards* for adding such details.

By attaching guards to paths at an OR-fork junction (when needed), we can solve most of the non-determinism problems associated to choices and unfeasible paths. One of the different alternatives would be chosen according to previous information set by tags.

These tags and guards should not be mandatory. They should be used only when required on specific paths. Non-determinism can still be present, if we do not have the information to solve it. In this way, a map can be incrementally extended if desired, without changing what is already there.

We give a simple example with tags and guards in figure 18. When a token takes (in a non-deterministic way) the upper path at the first OR-fork, a tag *T* is set to the value *Up*. Then, at the second OR-fork, it is forced to take the upper path again because a guard constrains the lower path to tags *T* different from the value *Up*. If the token follows the lower path at the first OR-fork, the tag *T* becomes *Down*, and the token can take either path at the second OR-fork, because it satisfies the guard of the lower path and there is no constraint associated to the upper path.

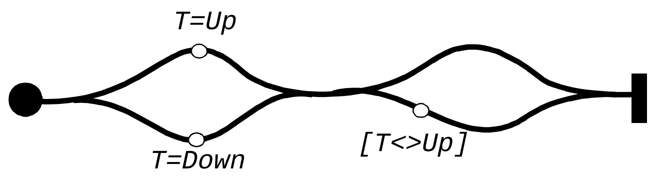


Figure 18: Use of tags and guards

This mechanism is implementable in LOTOS using an abstract data type *Tag* that enumerates possible tags and defines equality and inequality operations. Assignation of a value to a tag is done using the LOTOS **`let`** construct, and the guards are mapped onto LOTOS guards. A more complete description of the mapping is presented in section 4.5.5.

A timethread tool could implement many facilities to create and manipulate tags, in a user-friendly way. Such tool could also have the options to show or hide guards and/or tags to make the diagram simpler.

4.1.4 Tag Flow

The choice of a specific timethread path can influence the choice of another timethread path as an effect. Tags and interactions between these two timethreads are needed. There are only two ways of indicating inter-timethread interactions: preconditions and explicit interactions. Most timethread patterns are handled clearly by one or the other method. However, data has to flow at an interaction point for tag information to be transferred from one timethread to another. This can be implemented by *message passing* or by *global variables*.

Timethreads allow the use of both message passing and global variables. Since we are designing distributed systems, often without shared memory to manage such variables, global variables could be considered useless or dangerous to use. However, at a high level of abstraction, they can help designers delaying many decisions related to implementation while making the big picture clearer.

In this thesis, we formalize message passing only, because this way of handling data relates more closely to distributed systems in general. LOTOS offers the powerful mechanism of multi-way synchronization, to allow message passing (ADTs) at interaction points. Timethreads interactions can be mapped onto LOTOS multi-way synchronizations, and tag information can then flow from one timethread to another, or from the environment to a timethread.

In figure 19, there are three tags (LOTOS value identifiers) *C*, *T*, and *P*. There is a flow of information going from the environment to *C*, as expressed by an arrow going towards the timethread (incoming arrow). This information is associated to a token which takes a path according to the guards. Then *T* is set and passed to the next timethread (outgoing arrow), which accepts this value in a local tag *P* (incoming arrow). Finally, *P* is used to determine which path is to be taken in the second timethread.

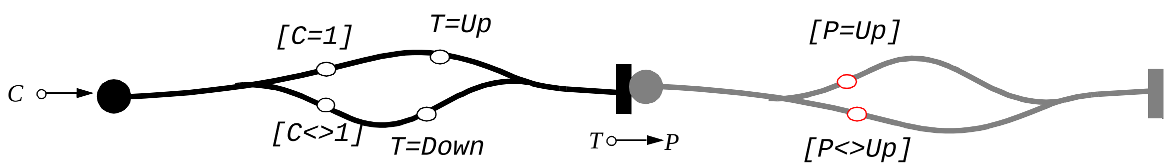


Figure 19: Flow of information

The tag flow symbol (○⟶) is not part of the traditional Timethread notation, although an equivalent data flow symbol has been used in the literature. There is a trade-off between capturing every detail and making the big picture clear. Too much notation clutters the picture. Nevertheless, we consider the tag flow notation to be simple and clear, and we will use it in this thesis to express flow of information between timethreads. We also use one type of data (*Tag*) only, in order for the mapping to be simple. At the abstraction level that interests us, timethread maps do not contain other types of data.

4.2 STDL Grammar

We use a *Single-Timethread Description Language* (*SDTL*), expressed as a *grammar,* to describe valid single timethreads. We use STDL to map timethreads onto formal languages such as LOTOS. It is a step towards a general description model that is expected to become the internal representation of timethread maps in a design tool.

The decomposition method of a timethread map (§3.1) would output a LARG description of the interactions and also the description of each individual timethread in STDL. Then a “compiler” would take these descriptions and output a LOTOS process for each timethread. In section 3.1, this was called the mapping method.

4.2.1 Requirements

In order to have the functionality expressed above, the grammar should:

- *G1)* Be general enough for the generation of LOTOS processes, while being independent. Other formalisms, such as Petri nets, could be used as output languages of other mapping methods.
- *G2)* Reflect a complete single timethread instead of segments, since these are less meaningful and do not fully express a timethread’s intentions.
- *G3)* Produce readable descriptions, so that people can actually read them. This helps in the design, debugging and implementation of a tool. This grammar almost becomes a language by itself.
- *G4)* Ease timethread-to-timethread transformations.
- *G5)* Support tags and data flow.
- *G6)* Be adaptable, i.e., it should be easily modified or extended in order to suit special needs or special notations.
- *G7)* Avoid redundant constructors while keeping the intentions of the timethread.
- *G8)* Possibly be integrated in a more general description model where timethread interactions and visual details are also supported.

4.2.2 Achievements

The STDL grammar presented in the next section achieves most of the goals mentioned:

- *G1)* LOTOS code can be generated from SDTL (this is the topic of the next section). However, mapping methods for other formalisms (such as Petri nets) are still untested.
- *G2)* SDTL generally reflects a complete timethread and its intentions, not only some of its paths or segments.
- *G3)* SDTL descriptions are easy to read and understand.
- *G4)* This has not been verified yet, although we give a taste of transformations in the next chapter.
- *G5)* Tags, guards and message passing are supported by SDTL.
- *G6)* We can adapt SDTL in order to include new special symbols or constructs, mostly by modifying the `<Seg>`, <`GenOptions`>, and `<WPOptions>` rewrite rules.
- *G7)* Only a few constructors can be considered as redundant (`Par` and `AndFork`, `Choice` and `OrFork`), and this trade-off aims at preserving timethread intentions.
- *G8)* SDTL can be associated to LARGs to cover interactions (§4.4). The integration of composition rules and visual informations to complete map description and representation is still work to be done.

4.2.3 SDTL Grammar in EBNF

The following context-free grammar represents the Single-Timethread Description Language. We use an Extended Backus-Naur Form where:

- Rewrite rules are of the form Left-Hand Side = Right-Hand side
- Non-terminal symbols are delimited by < and > as in `<Name>`
- Terminal symbols are in bold-italic as in ***`Name`***
- Alternative rules are separated by vertical bars (`|`) as in `<Delayed> | <Time>`
- Optional items are enclosed in square brackets (`[` and `]`) as in `[<RecTagValues>]`

- Optional lists, possibly empty, are enclosed by braces `{` and `}` as in `{<Seg>}`. A star (`*`) is added to indicate non-empty lists as in `{<Seg>}*`.
- Enumerations are expressed with `..` as in ***a..z***
- Comments are between `(*` and `*)`.

```
(* STDL, June 9, 1994 *)
(* Single-timethread definition. We separate stubs from complete timethread for future use. *)
<Timethread> =        Timethread <TTId> Is <StubOrTT> EndTT
<StubOrTT> =          <Stub> | <GenOptions> [<Internals>] <Trigger> <FirstPath>

(* Stub definition. No general options nor segments. They represent timethread stubs *)
<Stub> =              Stub <Trigger> <Result> EndStub

(* General options available to timethreads; can be extended. At the moment, there are two *)
(* non-exclusive options available for aborted and constrained timethreads. *)
<GenOptions> =        [<Aborted>] [<Constrained>]
<Aborted> =           AbortedOn ( <EventId> )
<Constrained> =       Constrained

(* A list of activities can be internal, i.e. hidden from the timethread’s environment. *)
<Internals> =         Internal <Identifier> {, <Identifier>}

(* A trigger has access to waiting places options and it might receive tag values. *)
<Trigger> =           Trigger <WPOptions> ( <TriggerId> [<RecTagValues>] )

(* The first path of a timethread does not need the keywords Path and EndPath to be clear. *)
<FirstPath> =         {<Seg>} <Result>

(* A result can send tag values to the environment or other interacting timethreads. *)
<Result> =            Result ( <ResultId> [<SendTagValues>] )

(* Waiting places options available; the list can be extended. At the moment, we consider two *)
(* exclusive options. *)
<WPOptions> =         [<Delayed> | <Timed>]
<Delayed> =           Delayed
<Time> =              Time

(* Types of segments available; the list can be extended *)
<Seg> =               <Abort> |
                      <Action> |
                      <AndFork> |
                      <Async> |
                      <Choice> |
                      <Loop> |
                      <Loss> |
                      <OrFork> |
                      <Par> |
                      <SegStub> |
                      <Sync> |
                      <Tag> |
                      <Waiting>

(* This event aborts another timethread. *)
<Abort> =             Abort ( <EventId> )

(* Indicates an action and its identifier. *)
<Action> =            Action ( <ActionId> )
```

```
(* Asynchronous/syncronous waiting places. Async can send tag values in passing, and Sync can *)
(* receive them. *)
<Async> =               Async ( <EventId> [<SendTagValues>] )
<Sync> =                Sync ( <EventId> [<RecTagValues>] )

(* Choice (OR-Fork & OR-Join) and OR-Fork (no join) segments. The Choice has at least two *)
(* optional list of segments, and the OrFork as a choice between the continuation of the *)
(* original path and at least one new path. Guards are optional. *)
<Choice> =              Choice [<Guard>] {<Seg>} {Or [<Guard>] {<Seg>} }* EndChoice
<OrFork> =              OrFork [<Guard>] Continue {Or [<Guard>] <Path>}* EndOrFork

(* A new path is a list of segments with a result. *)
<Path> =                Path {<Seg>} <Result> EndPath

(* A Loop is composed of two sections, Compulsory and Optional, indicated by their coresponding *)
(* keywords (they can be abbreviated by Comp and Opt). Guards are optional. *)
<Loop> =                Loop <LoopComp> <LoopOpt> EndLoop
<LoopComp> =            <CompSymb> [<Guard>] {<Seg>}
<LoopOpt> =             <OptSymb> [<Guard>] {<Seg>}
<CompSymb> =            Comp | Compulsory
<OptSymb> =             Opt | Optional

(* Loss of an instance or token. Can be guarded. *)
<Loss> =                Loss ( [<Guard>] <LossId> )

(* Par (AND-Fork & AND-Join) and AND-Fork (no join) segments. The Par has at least two *)
(* optional list of segments, and the AndFork adds at least one concurrent path. *)
(* Guards are forbidden (it is not a choice). *)
<Par> =                 Par {<Seg>} {And {<Seg>}}* EndPar
<AndFork> =             AndFork <Path> {And <Path>} EndAndFork

(* Segment stub definition. They represent path stubs. *)
<SegStub> =             SegStub ( <SegStubId> )

(* Waiting place that waits for an environment stimulus. *)
<Waiting> =             Wait <WPOptions> ( <EventId> [<RecTagValues>] )

(* Tags definition and tags passing (send and receive). *)
<Tag> =                 Tag ( <TagId> = <ValueId> )
<RecTagValues> =        ? <TagId> [<RecTagValues>]
<SendTagValues> =       ! <TagId> [<SendTagValues>]

(* Guard expressions. The list of equation operators and boolean operators can be extended. *)
(* These operators are currently based on LOTOS boolean and natural ADTs. *)
<Guard> =               Guard ( <GuardExpr> )
<GuardExpr> =           <TagId> <EqOp> <ValueId> |
                        not ( <GuardExpr> ) |
                        ( <GuardExpr> ) <BoolOp> ( <GuardExpr> )
<EqOp> =                eq | ne
<BoolOp> =              and | or | xor | implies | iff

(* Different identifiers used. They all start with a letter, except ValueId. *)
<ActionId> =            <Identifier>
<EventId> =             <Identifier>
<LossId> =              <Identifier>
<ResultId> =            <Identifier>
<SegStubId> =           <Identifier>
<TagId> =               <Identifier>
<TriggerId> =           <Identifier>
<TTId> =                <Identifier>
<ValueId> =             {<Alphanum>}*
<Identifier> =          <Letter> {<Alphanum>}
```

```
<Alphanum> =           <Digit> | <Letter>
<Digit> =              0..9
<Letter> =             a..z | A..Z
```

Appendix A presents the syntax diagrams of the STDL grammar. Rewrite rules are in rectangles and terminal symbols are in ellipses. Section 4.6 also presents the mappings from STDL to LOTOS according to these rules.

We believe SDTL is a step towards the automated generation of LOTOS specifications from timethread maps. Section 7.2 discusses more in depth the utility and advantages of this grammar. The next section presents most common cases of mapping from single timethreads to SDTL to LOTOS processes.

4.3 Single Timethreads in LOTOS

We present here the mapping method that generates LOTOS from single timethreads expressed in STDL. We mostly use level 1 specifications, although level 3 specifications are sometimes discussed. We also refer to the *Traveler System* introduced in section 3.3 to illustrate pertinent examples.

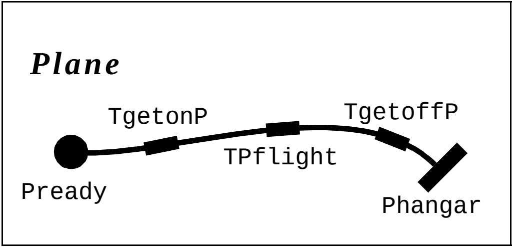
Section 4.3.1 presents basic timethread combinations, i.e. unconstrained and constrained starts, and the loop constructor. Section 4.3.2 shows the use of concurrent and alternate segments within a given timethread. In the upcoming examples, we present the timethread map, the corresponding STDL code, and then the resulting LOTOS process. We use a simpler LOTOS syntax (without gate parameters and process identification) in order to simplify the behaviour expressions generated. The complete LOTOS syntax is however respected in the appendices specifications.

Note that for some instances of basic combinations (e.g., constrained start and loop), the translation is not straightforward. However, we believe that it could be formally defined and automated, and this is why section 4.6 reviews most of the general mappings from STDL to LOTOS.

4.3.1 Basic Combinations

Sequence

The basic timethread of figure 20 has been already discussed in §4.1.2. The *sequence* is a very common pattern that can be found in the Traveler System (timethreads *Traveler* and *Plane*). In the following example, we consider the timethread *Plane* alone, without any interaction with *Traveler*. We obviously see that events and actions are directly mapped onto LOTOS gates.



```
Timethread Plane is
   Trigger (Pready)

   Action (TgetonP)
   Action (TPflight)
   Action (TgetoffP)
   Result (Phangar)
EndTT
```

```
Plane :=
   Pready;
   (
      TgetonP;
      TPflight;
      TgetoffP;
      Phangar; stop
   ) (* L1 *)
```

Figure 20: Example of sequence

To get a level 1 specification with recursion, we replace **`stop`** with the process instantiation `Plane[gates...]`. A level 3 specification is obtained by adding `||| Plane[gates...]` after **`stop`**. Generally, these are the only modifications needed to get a specification at a specific level. This is easily manageable for a tool or a compiler.

Internal Actions

In the previous timethread (fig. 20), we can observe all activities since nothing is declared internal. We defined events to be activities on which the environment or other timethreads interact, so they cannot be hidden or abstracted within a timethread, although whole interactions could be hidden at a higher level in a timethread map (see §5.2.1). Actions however can be internal to a timethread, and the STDL grammar allows this with the **`Internal`** construct.

For instance, we can make actions *TgetonP* and *TgetoffP* internal to the timethread (fig. 20). All actions declared internal is mapped onto a LOTOS hidden gate.

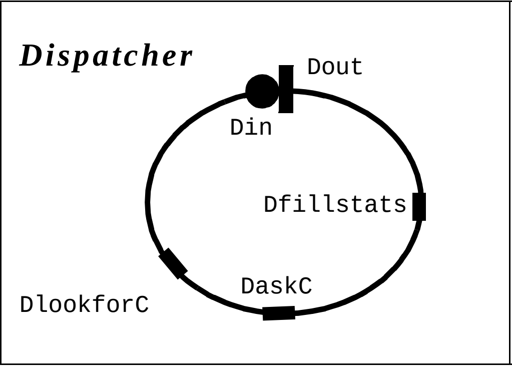
```
Timethread Plane is
   Internal
      TgetonP,TgetoffP
   Trigger (Pready)

   Action (TgetonP)
   Action (TPflight)
   Action (TgetoffP)
   Result (Phangar)
EndTT
```

```
Plane :=
   hide TgetonP,TgetoffP in

   Pready;
   (
      TgetonP;
      TPflight;
      TgetoffP;
      Phangar; stop
   ) (* L1 *)
```

Figure 21: Example of sequence with internal actions

A tool could allow a designer to select which actions should be internal to their respective timethreads. Note that we will not use internal actions in the remaining examples of this chapter. The *Traveler* and *Telepresence* systems will however use this very useful notion.

Constrained Start



```
Timethread Dispatcher is
   Constrained
   Trigger (Din)

   Action (DlookforC)
   Action (DaskC)
   Action (Dfillstats)
   Result (Dout)
EndTT
```

```
Dispatcher :=

   Din;
   (
      DlookforC;
      DaskC;
      Dfillstats;
      Dout; stop
   ) (* L1 *)
```

Figure 22: Example of constrained start

Figure 22 presents the *constrained start* timethread *Dispatcher*, without the loop and the interactions found in the original timethread map. The system allows only one instance (or one token) of timethread *Dispatcher* at a time, i.e., *Dispatcher* has to terminate for a new instance to start. For level 1 specifications (with or without recursion), there is no difference between a constrained start and a sequence, because there is at most one instance at a given time in both cases. However, we would like the triggering event not to be refused for level 3 specifications while an instance is executed. Those triggering events have to be accumulated in some way. For that purpose, the start waiting place needs internal machinery to manage the incoming of possibly many triggering events *Din*, while allowing only one token to go at a time.

The constrained start timethread *Dispatcher*, including a waiting place with necessary internal machinery, is specified at level 3 in specification 2:

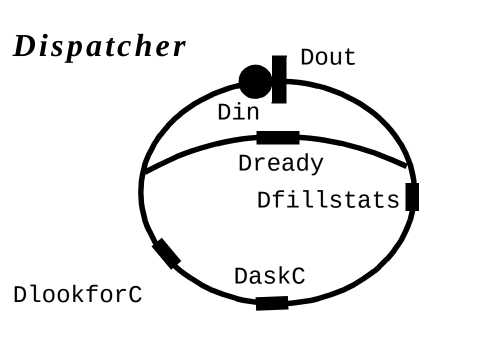
```
Dispatcher :=
   hide SyncCS in         (* hidden gate *)
      WP_CS |[SyncCS]| DispatcherSub
   where

   WP_CS :=               (* Waiting Place Machinery *)
      Din; (SyncCS; stop  (* Allows one token to go *)
            |||
            WP_CS)        (* Accumulation of Din *)

   DispatcherSub :=       (* Rest of the timethread *)
      SyncCS;
      DlookforC;
      DaskC;
      Dfillstats;
      Dout; DispatcherSub (* L3 *)
      (* Ready for the next token *)
```

Specification 2: Level 3 constrained start

We need two sub-processes, one for the waiting place (*WP_CS*) and one for the rest of the timethread (*DispatcherSub*), to represent the constrained start. These processes are synchronized on a hidden gate called *SyncCS*. We discuss waiting places with internal machinery in chapter 7.

Loop

The timethread *Dispatcher* includes a *loop* to indicate that someone can dispatch more than one cab. Figure 23 shows this loop, but we still ignore interactions for the moment. In the LOTOS representation of a loop, we have to define a sub-process (here *Dloop*) corresponding to the loop part and the rest of the timethread. For this purpose, we defined, in STDL, a *compulsory* segment **`Compulsory`** (must be executed at least once) and an *optional* segment **`Optional`** (may or may not be executed).



```
Timethread Dispatcher is
   Constrained
   Trigger (Din)
   Loop
     Compulsory
        Action (DlookforC)
        Action (DaskC)
        Action (Dfillstats)
     Optional
        Action (Dready)
   EndLoop
   Result (Dout)
EndTT
```

```
Dispatcher :=
   Din; (Dloop)
   where
   Dloop :=
      (* Compulsory Seg *)
      DlookforC;
      DaskC;
      Dfillstats;
      (
         (* Optional Seg *)
         Dready; Dloop
         []
         (* Exit loop *)
         Dout; stop
      ) (* L1 *)
```

Figure 23: Example of loop

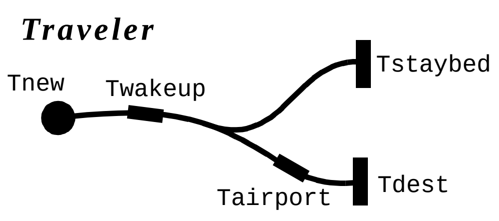
Again, recursion can be added to this process if necessary. Level 3 specifications are also possible: we simply replace `Din; Dloop` with `Din; (Dloop ||| Dispatcher)`. Lastly, guards can be attached to determine whether we loop again or not.

4.3.2 Concurrent and Alternate Segments

The *Traveler System* does not have any of the four types of concurrent and alternate segments (see §2.1.1). We will present short examples adapted from the *Traveler* timethread to illustrate these segments.

OR-Fork

Suppose the traveler has a choice between going to its destination and staying in bed (*Tstaybed*) that particular morning. These are two exclusive paths that will never join. Figure 26 presents such a (simplified) timethread. The LOTOS choice operator (`[]`) is used in the interpretation of the OR-Fork.



```
Timethread Traveler is
   Trigger (Tnew)
   Action (Twakeup)
   OrFork
   (* Continued segment *)
       Continue
     Or
   (* New path *)
       Path
          Result (Tstaybed)
       EndPath
   EndOrFork
   Action (Tairport)
   Result (Tdest)
EndTT
```

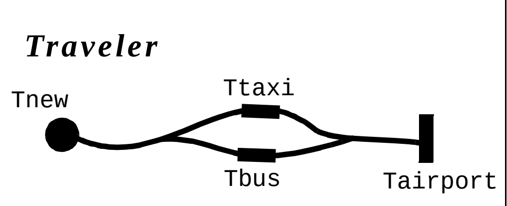
```
Traveler :=
   Tnew;
   (
      Twakeup;
      (
         (* Continued seg *)
         Tairport;
         Tdest; stop
       []
         (* New path *)
         Tstaybed; stop
      )
   ) (* L1 *)
```

Figure 24: Example of OR-Fork

In STDL, the choice construct allows multiple alternatives (more than two options) and this is reflected in the LOTOS code accordingly. When an **`OrFork`** occurs, we choose between the continuation of the timethread (**`Continue`**) and new path segments (**`Path`**). The latter have their own resulting events. As expressed in the STDL grammar, at least one new segment is needed in the **`OrFork`** construct. We also have the possibility to add guards to the all segments. Finally, we generate recursive specifications by modifying the **`stop`** of each alternative, and level 3 specifications by adding `||| Traveler` before the last parenthesis, as in the previous section.

Choice

The *choice* is the combination of an OR-Fork with an OR-Join. It indicates that a token can follow one of many different paths for a while, but all these paths merge in a common path later. For instance, our traveler still wants to get to the airport, but she now has a choice between a taxi (*Ttaxi*) and a bus (*Tbus*). This case is presented in figure 25.



```
Timethread Traveler is
   Trigger (Tnew)

   Choice

       Action (Ttaxi)
     Or
       Action (Tbus)
   EndChoice

   Result (Tairport)
EndTT
```

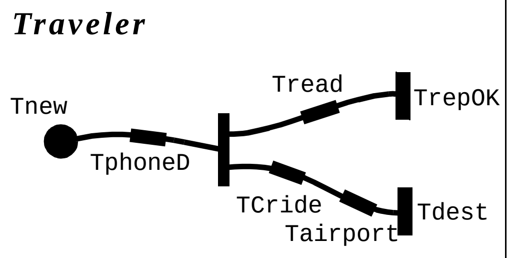
```
Traveler :=
   Tnew;
   (
      hide SyncOR in
      (
         Ttaxi; SyncOR; stop
       []
         Tbus; SyncOR; stop
      )
      |[SyncOR]|
      SyncOR;
      Tairport; stop
   ) (* L1 *)
```

Figure 25: Example of choice (OR-Fork & OR-Join)

Again, we have added a hidden gate *SyncOR* to synchronize the end of the choice with the rest of the timethread. Multiple choices are supported by STDL and LOTOS, and guards can be added.

AND-Fork

The traveler is able to do many things concurrently. In the next example (fig. 26), the traveler has a report to read. After she phones the dispatcher, she can read this report (*Tread*) before her taxi ride, before getting to the airport, before or after the arrival to the destination. This is represented with two concurrent paths after a AND-Fork.



```
Timethread Traveler is
   Trigger (Tnew)
   Action (TphoneD)
   AndFork
     (* New path *)
     Path
       Action (Tread)
       Result (TrepOK)
     EndPath
   EndAndFork
   (* Continued segment *)
   Action (TCride)
   Action (Tairport)
   Result (Tdest)
EndTT
```

```
Traveler :=
   Tnew;
   (
      TphoneD;
      (
         (* New path *)
         Tread
         TrepOK; stop
       |||
         (* Continued seg *)
         TCride;
         Tairport;
         Tdest; stop
      )
   ) (* L1 *)
```

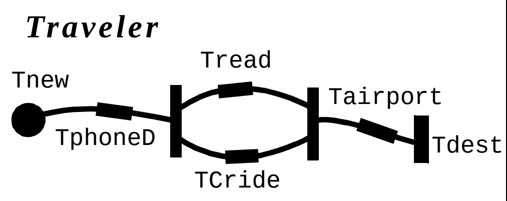
Figure 26: Example of AND-Fork

The LOTOS interleaving operator is used here to represent that two tokens follow the two paths concurrently. The **`AndFork`** adds new concurrent path segments, and we may have more than two exiting paths (thus at least one new **`Path`** segment), without guards.

Par

Sometimes, we do not care in what order some activities occur, or we want to represent them as occurring concurrently. A constructor named *par*, which is an AND-Fork followed by a AND-Join, is used in figure 27 to express such a concept. In this example, the traveler has to read her report after she phones the dispatcher and before she arrives at the airport. However, she can read it before or after her taxi ride (or during her ride, if we interpret the interleaving as true concurrency).

The STDL **`Par`** construct is mapped onto the LOTOS parallel composition operator. The synchronization between all concurrent paths occurs on a hidden gate (*SyncAND*). As for the other constructors, we can specify more than two concurrent paths, and we can generate recursive or level 3 specifications.



```
Timethread Traveler is
   Trigger (Tnew)
   Action (TphoneD)

   (* Concurrent segs *)
   Par
       Action (Tread)
     And
       Action (TCride)
   EndPar

   (* Rest of Traveler *)
   Action (Tairport)
   Result (Tdest)
EndTT
```

```
Traveler :=
   Tnew;
   (
      TphoneD;
      (
         hide SyncAND in
         (
            Tread;
            SyncAND; stop
          |[SyncAnd]|
            TCride;
            SyncAND; stop
         )
         |[SyncAnd]|
         SyncAnd;
         Tairport;
         Tdest; stop
      )
   ) (* L1 *)
```

Figure 27: Example of par (AND-Fork & AND-Join)

The reason for using a hidden gate instead of the **`exit`** followed by the enable operator (>>) to synchronize concurrent segment is to preserve consistency with other types of interactions (see §4.4.1).

4.4 Timethread Interactions

A timethread map is a collection of interacting timethreads. We can differentiate two types of interactions: timethread starting (§4.4.1), where one or many timethreads start one or many other timethreads, and synchronous/asynchronous interactions along timethread paths (§4.4.2). The structural part of these interactions will be obtained using the timethread decomposition method and the LAEG method [Bor 93].

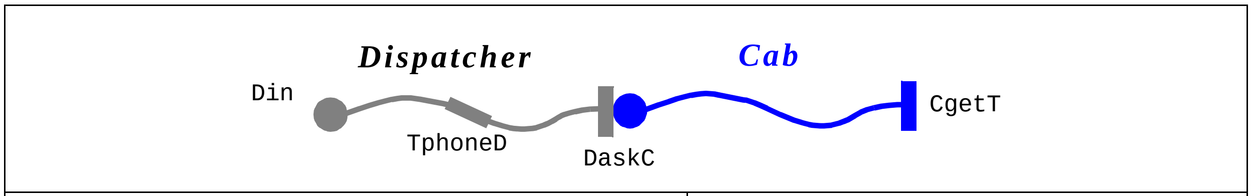
The decomposition method aims at mapping the topology of interacting timethreads onto a LARG and generating descriptions of individual timethreads in STDL. Section 4.3 showed how STDL descriptions are obtained from individual timethread. The current sections covers the mapping of timethread maps onto LARGs, and the generation of LOTOS structural expressions using the LAEG method.

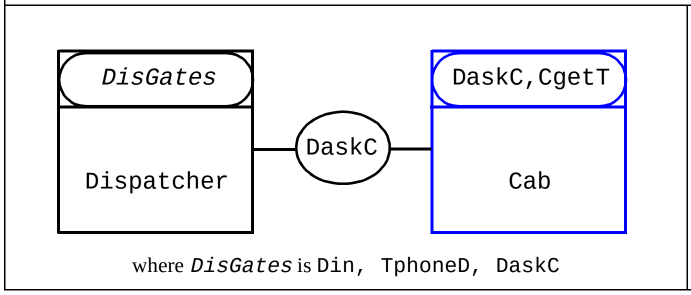
4.4.1 Starting Concurrent Timethreads

This section proposes several basic scenarios, related to the *Traveler System*, where timethreads are started. Their goal is to present interactions constructors, which are represented as juxtapositions of basic symbols (refer to §2.1.2).

Concatenation

The *concatenation* is probably the most simple interaction, and several basic options are discussed here. In the example of figure 28, the dispatcher receives a phone call and asks for a cab. This enables a cab to go and get a traveler (*CgetT*).





```
Dispatcher [Din, TphoneD, DaskC]
|[DaskC]|
Cab [DaskC, CgetT]

where

   (* Process definitions *)
```

Figure 28: Example of concatenation

The mapping onto a LARG is given. There are two processes, one for each timethread, synchronized on gate *DaskC*. This gate could be potentially hidden at a higher lever of abstraction (see §5.2.1 for further explanations on internal events). The LAEG method generates the corresponding LOTOS structural expression. The analysis step does not find any ambiguity in the LARG, and the expression shown in the figure is directly generated.

The process definitions are given below (fig. 30). They correspond to two sequences where the result of the first timethread (*Dispatcher*) is the same event as the triggering event of the second one (*Cab*). The synchronization occurs on that specific event (*DaskC*).

```
Timethread Dispatcher is
   Trigger (Din)
   Action (TphoneD)
   Result (DaskC)
EndTT
```

```
Dispatcher :=
   Din;
   (
      TphoneD;
      DaskC; stop
   ) (* L1 *)
```

```
Timethread Taxi is
   Trigger (DaskC)
   Result (CgetT)
EndTT
```

```
Taxi :=
   DaskC;
   (
      CgetT; stop
   ) (* L1 *)
```

Figure 29: Level 1 process definitions of figure 28

We can use different levels of specification for these two processes. In figure 30, we used level 3 specifications as process definitions.

```
Dispatcher :=
   Din;
   (
      TphoneD;
      DaskC; stop
      |||
      Dispatcher
   ) (* L3 *)
```

```
Taxi :=
   DaskC;
   (
      CgetT; stop
      |||
      Dispatcher
   ) (* L3 *)
```

Figure 30: Level 3 process definitions of figure 28

Gate Synchronization vs Enable Operator

In figure 28, we used synchronization on an explicit gate instead of the LOTOS enable operator (>>) in order to avoid problems associated to levels of specification. An instance of such problems is given in figure 31, where a timethread *Q* is concatenated to the end of a timethread *P*:

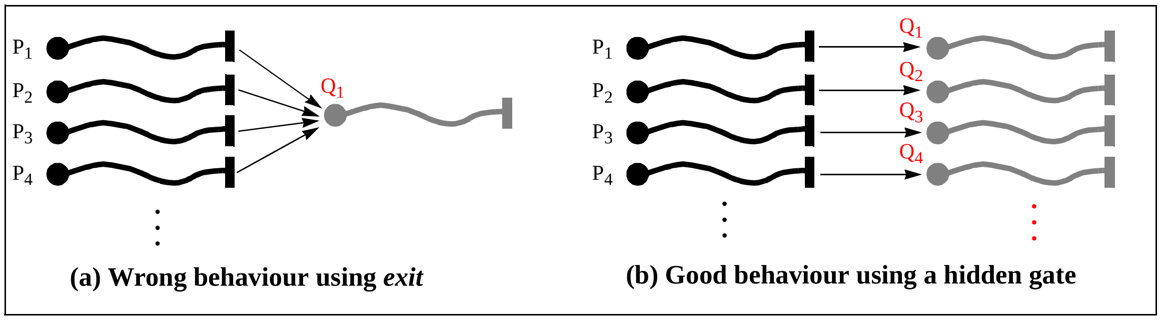

P1
P2
P3
P4
Q1
(a) Wrong behaviour using exit
P1
P2
P3
P4
Q1
Q2
Q3
Q4
(b) Good behaviour using a hidden gate

Figure 31: LOTOS synchronization on a concatenation

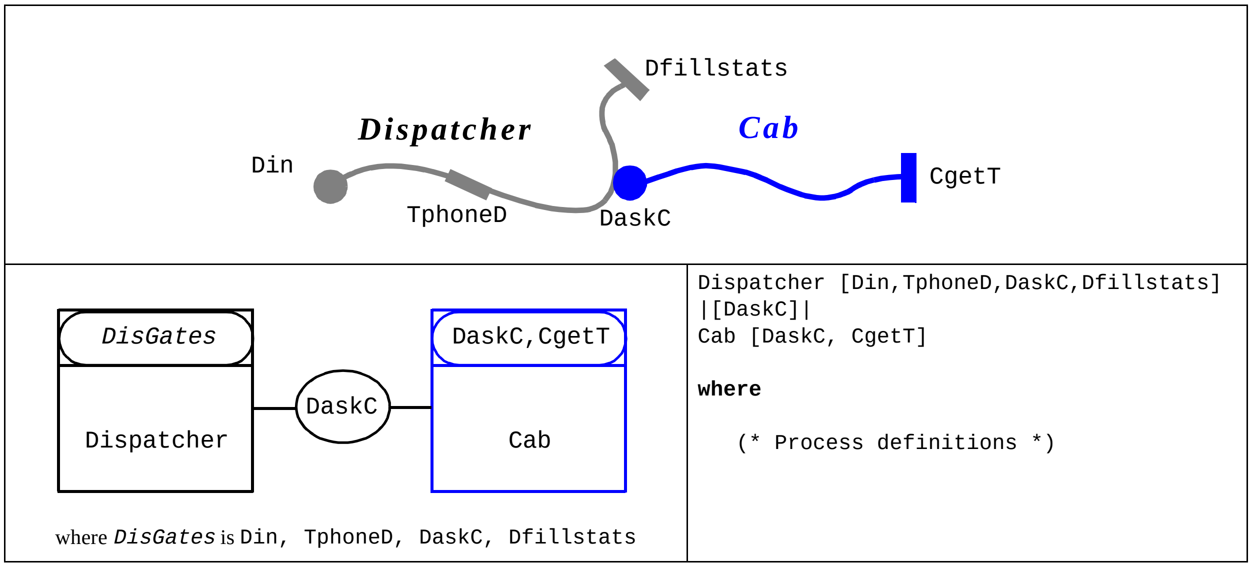
With a level 3 specification, we cannot use the LOTOS enable operator (>>) to describe the interaction because all instances of *P*, although they are executed in parallel, would have to synchronize on ***exit*** before *Q* starts (see fig. 31a). This is impossible because there would be an infinite number of instances of *P*. By using synchronization on a potentially hidden gate, an instance of process *Q* is created every time an instance of process *P* terminates (fig. 31b), whitout any problem.

We considered the *generalized termination* (>e_i>), presented in [QuA 92], as an alternative to synchronization on potentially hidden gates. However, although the problem of the standard enable operator is solved, the new operator was found to act like a disabling (all instances of *P* would be destroyed after an enabling). Also, the extension of [QuA 92] needs some changes in the underlying model of LOTOS (a new compound event), and their proposal is not standardized yet. Because of these reasons, this option was rejected.

The use of potentially hidden gates leaves us with fewer LOTOS operators to consider. This satisfies one of the guiding rules presented in §4.1.1.

In-Passing Start

We adapt here the previous example. The concatenation is replaced with an *in-passing start*, and the dispatcher fills his statistics at the end (see fig. 32).



```
Dispatcher [Din,TphoneD,DaskC,Dfillstats]
|[DaskC]|
Cab [DaskC, CgetT]

where

   (* Process definitions *)
```

Figure 32: Example of in-passing start

This interaction is asynchronous, i.e, *Dispatcher* does not wait at the interaction point, but *Cab* does. The LARG model expresses synchronous interactions only, and thus the “asynchronous” part must be simulated in the processes themselves. We decided to express it in the description of the non-waiting timethread *Dispatcher* as shown in figure 33:

```
Timethread Dispatcher is
   Trigger (Din)

   Action (TphoneD)
   (* Async event *)
   Async (DaskC)
   (* Rest of the path *)
   Result (Dfillstats)

EndTT
```

```
Dispatcher :=
   Din;
   (
      TphoneD;
      (
         DaskC; stop
         |||
         Dfillstats;
         stop
      )
   ) (* L1 *)
```

```
Timethread Cab is
   Trigger (DaskC)

   Result (CgetT)
EndTT
```

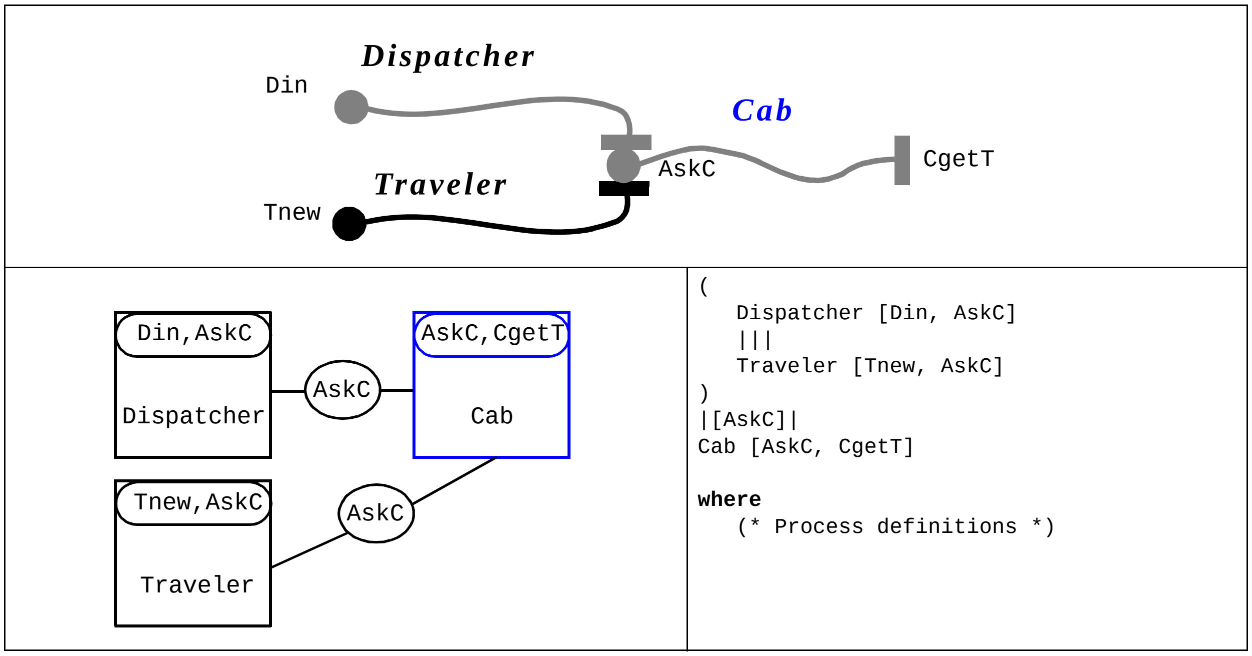
```
Cab :=
   DaskC;
   (
      CgetT; stop
   ) (* L1 *)
```

Figure 33: Process definitions of figure 32

We prefer this way of simulating asynchronous interactions to the insertion of a buffer between two processes. The latter option causes another process to be created and the asynchronous event to be split (*DaskC* would become *DaskCSend* and *DaskCReceive*). Our option is much simpler and only one event is necessary.

OR-Start

Many-to-one timethread interactions are possible. In this example (fig. 34), we suppose that either a *Dispatcher* or a *Traveler* can ask for a *Cab*.



```
(
   Dispatcher [Din, AskC]
   |||
   Traveler [Tnew, AskC]
)
|[AskC]|
Cab [AskC, CgetT]

where
   (* Process definitions *)
```

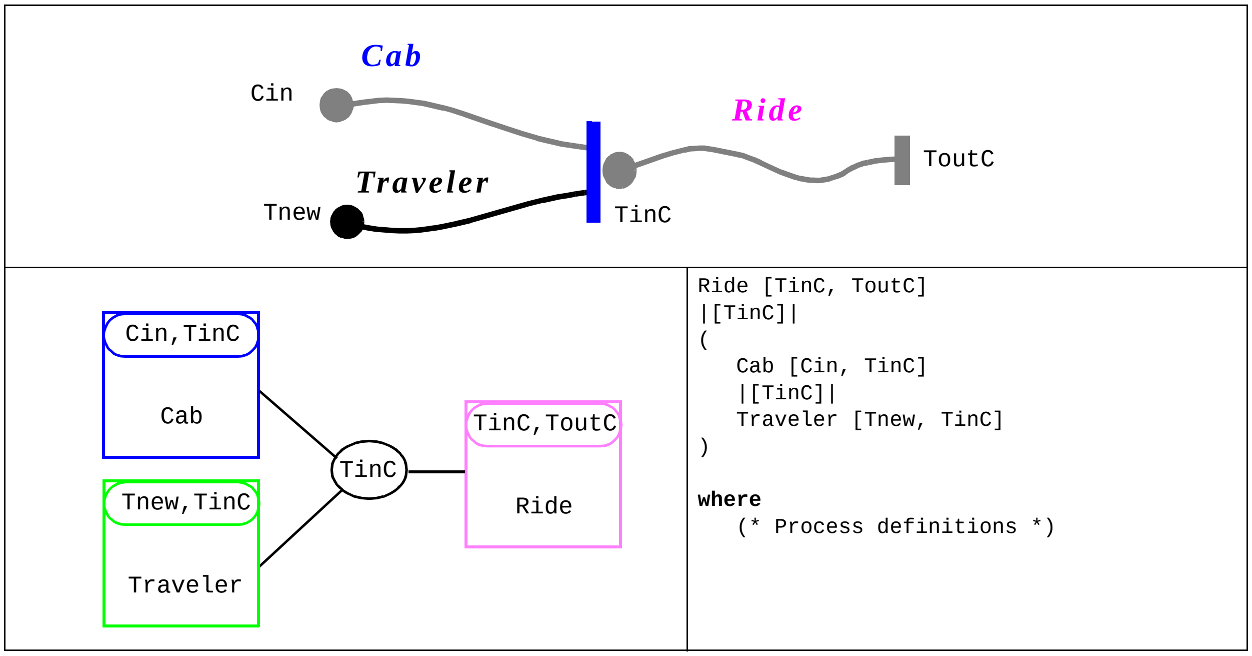
Figure 34: Example of OR-Start

The LAEG method finds *OR-Starts* to lead to structural ambiguities. LOTOS structural expressions cannot be directly derived from such LARGs. However, as explained in section 2.3.2, we apply the *structural ambiguity grouping* to solve this problem. The resulting LOTOS structural expression is presented in figure 34.

Process definitions are derived in the same way as the concatenation (thus, directly from the map). Different levels of specification lead however to different global behaviours. For instance, if the three processes involved are specified using level 1 without recursion, then once *Cab* has been triggered by one incoming timethread, the other timethread will deadlock on *AskC* because we only have one instance of *Cab*. If we use recursion, then the second incoming timethread has to wait for *Cab* to finish before synchronizing on *AskC*. There are no such problem at a level 3: *AskC* is always available to as many instances of incoming timethreads as possible. This is a big advantage of level 3 specifications over level 1 specifications, because we do not have to consider the non-availability of *AskC* while validating the map.

AND-Start

Suppose we need a token from a timethread *Traveler* and one from *Cab* to create a token on a timethread *Ride*, as in figure 35. The LARG representation of this *AND-Start* is a three-way rendezvous between the three processes. The LOTOS structural expression is easily generated from the LARG, and process definitions are generated as usual.



```
Ride [TinC, ToutC]
|[TinC]|
(
   Cab [Cin, TinC]
   |[TinC]|
   Traveler [Tnew, TinC]
)

where
   (* Process definitions *)
```

Figure 35: Example of AND-Start

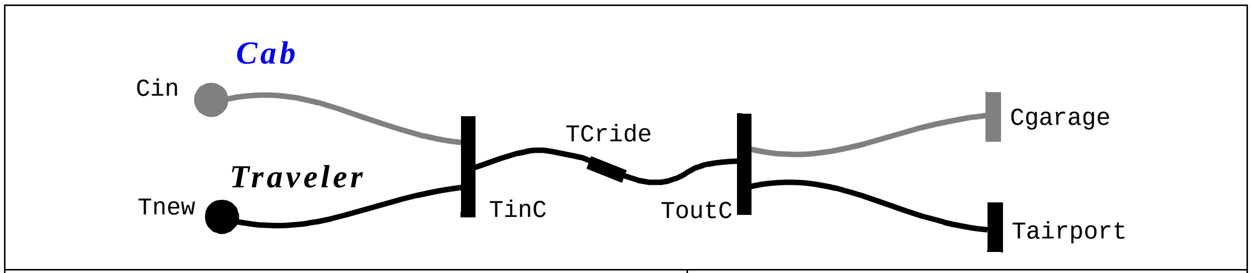
We saw that one-to-one starts and many-to-one starts can be mapped onto LOTOS. Any starting of timethreads, including one-to-many and many-to-many starts, are obtained by the same construction. We generate the LOTOS structural expressions in a similar way.

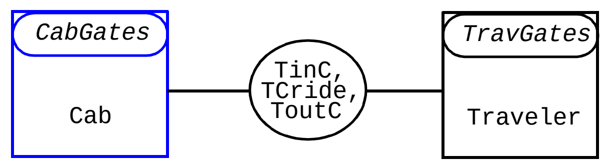
4.4.2 Synchronous / Asynchronous Interactions Along Paths

This section deals with different kinds of shared paths, synchronizations and triggering events between timethreads. Waiting places along timethreads will be used.

Synchronous Join-Fork

A traveler and a cab have to synchronize during their ride to the airport. They wait for each other on *TgetinC*, go together (*TCride*), and then go their separate ways after *TgetoutC*. This *synchronous Join-Fork* of timethreads *Traveler* and *Cab* is shown in figure 36:





where *CabGates* is `Cin, TinC, TCride, ToutC, Cgarage`
and *TravGates* is `Tnew, TinC, TCride, ToutC, Tairport`

```
Cab [Cin, TinC, TCride, ToutC, Cgarage]
|[TinC, TCride, ToutC]|
Traveler [Tnew, TinC, TCride, ToutC,
          Tairport]

where
   (* Process definitions *)
```

Figure 36: Example of synchronous Join-Fork

All activities on the common segment are put in the gate set linking the two processes. Therefore, these two processes are synchronized on all activities on this path segment. Since interactions occur all along this segment, actions cannot be hidden locally. In fact, actions such as *TCride* are more than actions; they are also interactions. The current interpretation results in identical copies of the common segment in the two timethreads description. However, a timethread tool user should not be bothered with such problem.

Identical copies lead to potential problems. One can be encountered in the case of modifications, which will have to be applied consistently to all copies. A second one relates to some well-known problems due to non-deterministic choices in the LOTOS world. We know that the composition of a process *P* with itself is not always equivalent to *P*. Figure 37 shows an example of a non-deterministic process *P* and the composition *P* || *P* where `P:= a; b; `**`stop`**` [] a; c; `**`stop`**. Deadlocks can result in the composition process *P* || *P* after the action *a*.

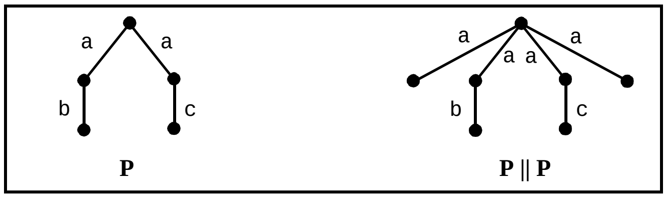
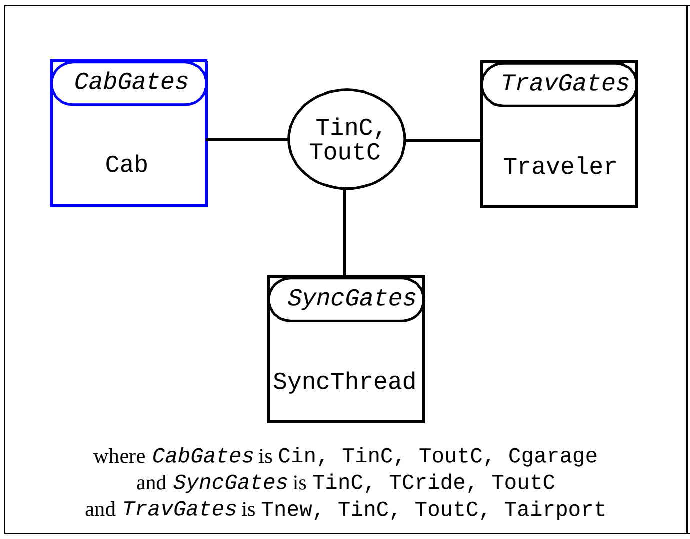


Figure 37: LTSs of P and P||P

Could such problem occur between two synchronizing timethreads? We believe it could not. Timethreads have unique identifiers for their activities, so a choice between identical actions is impossible. Also, because internal actions are not allowed in a synchronized segment (all actions have to be observable to synchronize), there will never be a non-deterministic choice due to internal actions. Therefore, we think these are sufficient conditions to avoid most problems with synchronized timethreads.

There is another way of formalizing the synchronous Join-Fork, presented in [Amy 93], that avoids all the problem due to identical copies (fig. 38). We introduce a new process called *SyncThread* that represents the common synchronized segment (here *TCride*). It is almost considered a timethread by itself, and could be specified at different levels. The processes synchronization is done on the first (*TinC*) and last events (*ToutC*).



where *CabGates* is `Cin, TinC, ToutC, Cgarage`
and *SyncGates* is `TinC, TCride, ToutC`
and *TravGates* is `Tnew, TinC, ToutC, Tairport`

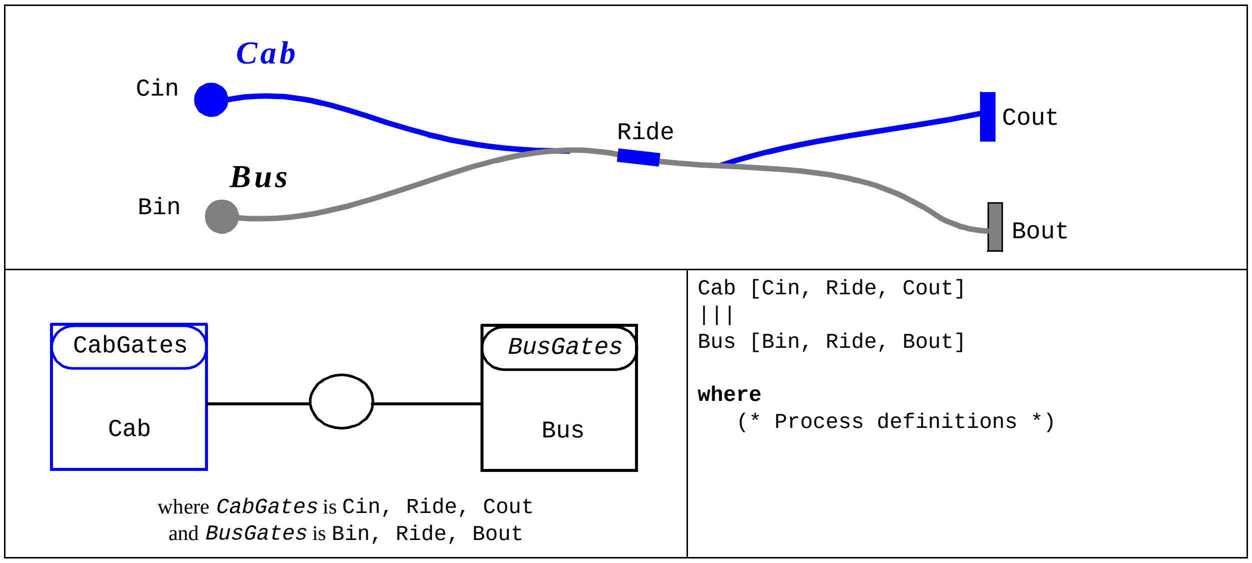
```
Cab [Cin, TinC, ToutC, Cgarage]
|[TinC, ToutC]|
SyncThread [TinC, TCride, ToutC]
|[TinC, ToutC]|
Traveler [Tnew, TinC, ToutC, Tairport]

where
   SyncThread :=
      TinC;           (* First event *)
      (
         TCride;      (* Sync. segment *)
         ToutC; stop  (* Last event *)
      )

   (* Process definitions of Cab and *)
   (* Traveler, without TCride. *)
```

Figure 38: Second interpretation of synchronous Join-Fork

This interpretation, similar to semaphores in LOTOS, could facilitate timethread transformations by eliminating the consistency checking between synchronizing processes. Also, we are able to hide internal actions in *SyncThread*, while this was impossible in the previous alternative because all actions had to be observable to synchronize. However, the cost is a supplementary process, which is not consistent with the one-to-one relation between timethreads and LOTOS processes, and increased complexity w.r.t. tag flow. In this thesis, we prefer to use the solution without additional processes to this alternate solution.

Asynchronous Join-Fork

The following example (fig. 39) presents a case where an *asynchronous Join-Fork* is required. A *Bus* and a *Cab* possess both the ability to perform the *Ride* action, without any synchronization. This simple example shows that the asynchronous Join-Fork is more a visual hint than an interaction between timethreads. This is why the gate set is empty.



where *CabGates* is `Cin, Ride, Cout`
and *BusGates* is `Bin, Ride, Bout`

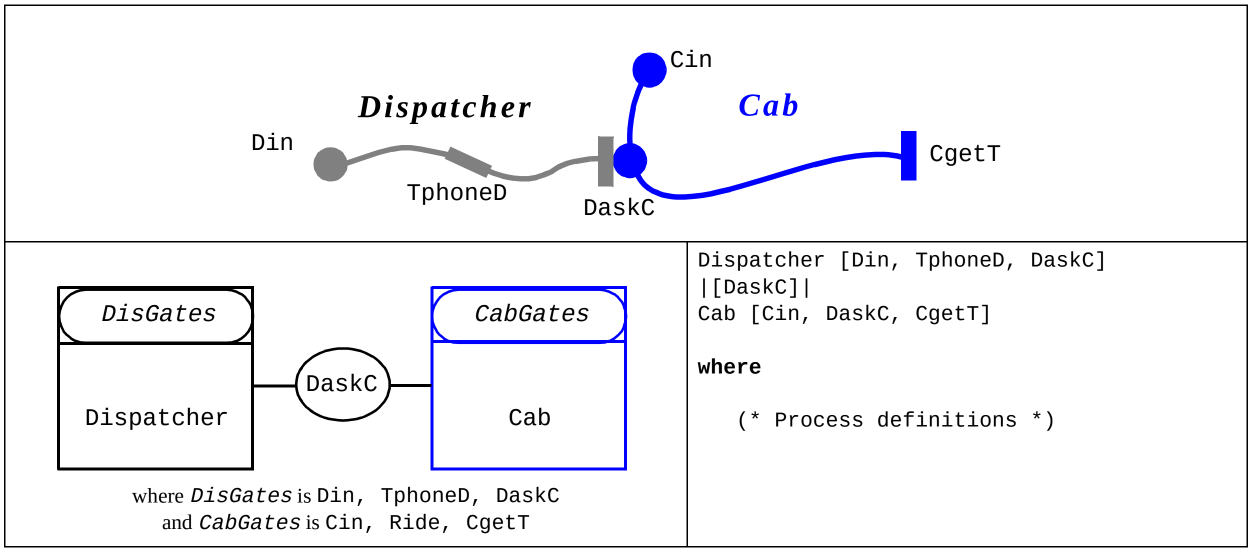
```
Cab [Cin, Ride, Cout]
|||
Bus [Bin, Ride, Bout]

where
   (* Process definitions *)
```

Figure 39: Example of asynchronous Join-fork

The interpretation is simply reduced to two interleaving processes having both the gate *Ride*.

End-Trigger

This example slightly differs from the concatenation (see fig. 28). In figure 40, *Cab* starts independently from *Dispatcher*, but it has to wait for *DaskC* before continuing. Since *DaskC* is the resulting event of *Dispatcher*, an *end-trigger* is required.



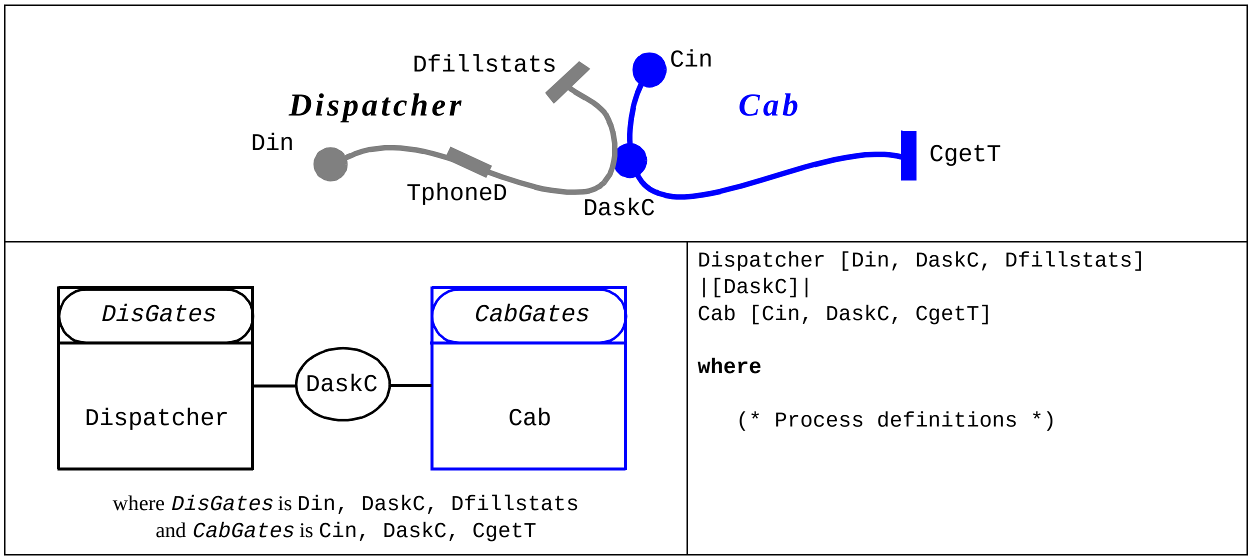
```
Dispatcher [Din, TphoneD, DaskC]
|[DaskC]|
Cab [Cin, DaskC, CgetT]

where

   (* Process definitions *)
```

Figure 40: Example of end-trigger

In STDL, the waiting place *DaskC* is represented with a `<Sync>` segment. Tags can flow from *Dispatcher* to *Cab*.

In-Passing-Trigger

We modify the previous example to insert an *in-passing-trigger*. The dispatcher, after having asked for a cab, does not wait for anything to happen and fills his statistics. Figure 41 illustrates this asynchronous interaction.



```
Dispatcher [Din, DaskC, Dfillstats]
|[DaskC]|
Cab [Cin, DaskC, CgetT]

where

   (* Process definitions *)
```

Figure 41: Example of in-passing-trigger

In STDL, the interaction point *DaskC* is represented as **`Async`**`(DaskC)` in *Dispatcher*, and **`Sync`**`(DaskC)` in *Cab*. The resulting process definitions are similar to the ones in figure 33.

Other Interactions

Complex interactions, involving more than two timethreads, are described using the same approach. A separate architectural approach (such as the LARG model) combined with single timethread description (in STDL) gives us the LOTOS architecture required to represent the most complex timethreads interactions.

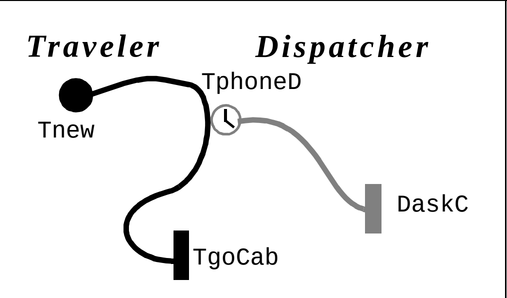
4.5 Special Symbols

In this section, we give a LOTOS interpretation to a selection of timethread special symbols (see §2.1.3). In order to keep the mappings simple, LARG representations and LOTOS structures are not given when they can be easily derived from the previous sections.

4.5.1 Timers

Delays

A timer symbol, used as a *delayed waiting place* triggering a timethread, expresses a *delayed start*. In figure 42, the dispatcher waits for a certain time before asking for a cab. In STDL, the keyword **`Delayed`**, an option in the grammar (`<WPOptions>`), is used to express such waiting places. This delay is interpreted in LOTOS as the internal action *Delay*.



```
Timethread Dispatcher is
   Delayed
   Trigger (TphoneD)
   Result (DaskC)
EndTT
```

```
Dispatcher :=
   hide Delay in
   Delay;
   TphoneD;
   (
      DaskC; stop
   ) (* L1 *)
```

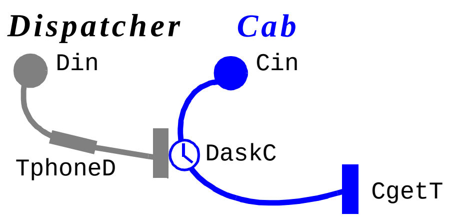
Figure 42: Example of delayed start

Time is an abstract notion in LOTOS. We think that an internal action is sufficient to clearly express a delay, and therefore no time extension is needed. In fact, strictly speaking, even an internal action is not necessary because, in LOTOS, every action implies arbitrary delay. However, an explicit internal action makes the specification clearer and more meaningful.

A delayed waiting place can also be used alone along a timethread path. It is interpreted in the same way, i.e, with an internal action *Delay*.

Timeouts

We also use timer symbols to express *time waiting places*. Assume a cab is waiting for the dispatcher’s orders. After a certain period of time (*TimeOut*), if the cab does not receive anything, it can decide to continue its way to get a traveler by his own. In figure 42, we use the STDL waiting place option **`Time`** to indicate such a time waiting place.



```
Timethread Cab is
   Trigger (Cin)
   Sync Time (DaskC)
   Result (CgetT)
EndTT
```

```
Cab :=
   Cin;
   (
     hide TimeOut, SyncTO in
     (
         TimeOut;
         SyncTO! TOut;
         stop
         []
         DaskC;
         SyncTO! OK;
         stop
      )
      |[SyncTO]|
      SyncTO ?ResultTO:Tag;
      CgetT; stop
   ) (* L1 *)
```

Figure 43: Example of time waiting-place

We map **`Time`** onto a LOTOS hidden gate *TimeOut*. A cab has a choice between waiting an order or continuing after *TimeOut*. Both choice lead to the internal event *SyncTO* for internal synchronization with the rest of the path. Note that we use a special tag (*ResultTO*) that represents the state of an instance (or a token). This tag, with a value of *TOut* or *OK*, can be used later to constrain alternatives with guards (see §4.5.5 for tags use). Also, time waiting places can be used as triggering events.

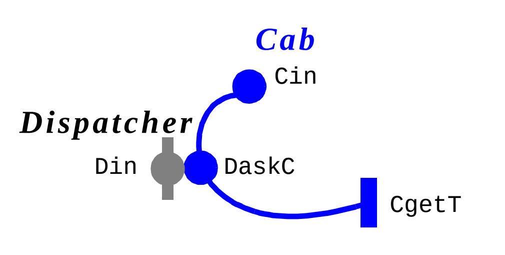
4.5.2 Stubs

Stubs represent abstract paths or timethreads. We distinguish two types of stubs:

- *Timethread stubs*: they represent an abstract complete timethread, including a trigerring event and a resulting event. They however do not include any segment. Timethread stubs usually interact with other timethreads. In STDL, we define them with the keyword **`Stub`** (see §4.2.3).

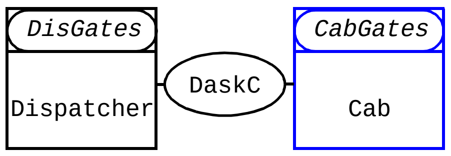
- *Path stubs*: they represent a missing or abstract path along a timethread. They are located directly on the body of a timethread. In STDL, we define them using the segment **`SegStub`**.

As a timethread stub example, the *Dispatcher* stub of figure 44 provides the *DaskC* event necessary for *Cab* to continue. The triggering event of this stub is *Din*, and nothing else is said about *Dispatcher.*



```
Timethread Cab is
   Trigger (Cin)
   Sync (DaskC)
   Result (CgetT)
EndTT
```

```
Cab :=
   Cin;
   (
      DaskC;
      CgetT; stop
   ) (* L1 *)
```



where *`DisGates`* is `Din, DaskC`
and *`CabGates`* is `Cin, DaskC, CgetT`

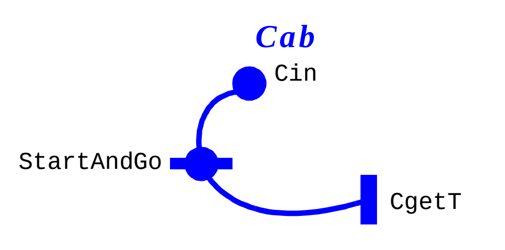
```
Name Dispatcher is
   Stub
      Trigger (Din)
      Result (DaskC)
   EndStub
EndTT
```

```
Dispatcher :=
   Din;
   (
      DaskC; stop
   ) (* L1 *)
```

Figure 44: Example of a timethread stub

The timethread stub is mapped onto a LOTOS process with only one triggering and one resulting event. As shown in the LARG, they can interact with other timethreads as if they were timethreads themselves.

Figure 44 presents an example of a timethread were the path stub *StartAndGo* abstracts a more complex path were the cab driver might check the fuel, start the car, go to work, etc .



```
Timethread Cab is
   Trigger (Cin)
   SegStub (StartAndGo)
   Result (CgetT)
EndTT
```

```
Cab :=
   Cin;
   (
      StartAndGo;
      CgetT; stop
   ) (* L1 *)
```

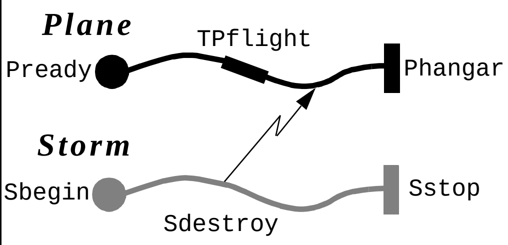
Figure 45: Example of a path stub

The path stub is simply mapped onto a LOTOS gate, in the same way as ordinary actions. We forbid any interaction on these stubs.

Stubs are mostly used ase visual cues for the moment. Eventually, they will be part of a model (and a tool) where they will be refined and will represent different levels of abstraction in a timethread map.

4.5.3 Abort

The abort operator and its LOTOS interpretation are presented in figure 46. Suppose a *Storm* timethread that destroys (*Sdestroy*) all instances of *Plane*, wherever they are. In STDL, the option **`AbortedOn`** shows that a timethread can be aborted, and the construct **`Abort`** shows the aborting event.

 Plane
TPflight
Pready
Phangar
Storm
Sbegin
Sstop
Sdestroy
	`Timethread` `Plane` **`is`**
	   `AbortedOn` `(Sdestroy)`
	   `Trigger` `(Pready)`
	   `Action` `(TPflight)`
	   `Result` `(Phangar)`
	`EndTT`
		`Plane :=`
		   `(`
		      `Pready;`
		      `(`
		         `TPflight;`
		         `Phangar;` **`stop`**
		      `)`
		   `)`
		  `[> Sdestroy;` **`stop`** *`(* L1 *)`*

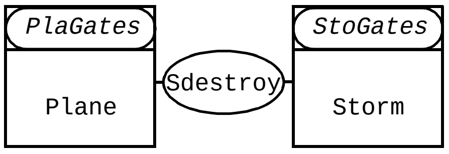
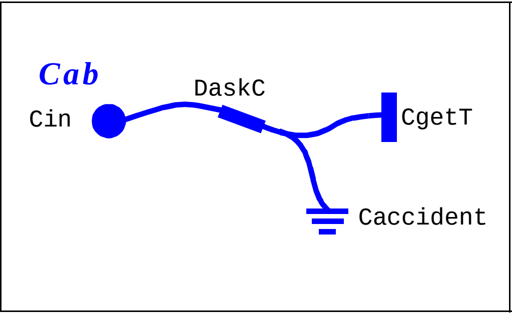
 where *`PlaGates`* is
`Pready, TPflight, Phangar, Sdestroy`
and *`StoGates`* is
`Sbegin, Sdestroy, Sstop`
	`Timethread` `Storm` **`is`**
	   `Trigger` `(Sbegin)`
	   `Abort` `(Sdestroy)`
	   `Result` `(Sstop)`
	`EndTT`
		`Storm :=`
		   `Sbegin`
		   `(`
		      `Sdestroy;`
		      `Sstop;` **`stop`**
		   `)` *`(* L1 *)`*

Figure 46: Example of an abort

We map the abort onto the LOTOS disabling operator (`[>`), and the aborting event (*Sdestroy*) becomes an interaction point between the two timethread. In this way, an abort really kills all instances of a timethread until a new instance is triggered. This special symbol must therefore be used with special care.

4.5.4 Loss

When we want to consider robustness in a timethread map, we often use a *Loss* symbol to express that a token can be lost along a timethread path. In figure 46, a cab can get lost (forever) after being asked by the dispatcher to take a traveler.



```
Timethread Cab is
   Trigger (Cin)
   Action (DaskC)
   Loss (Caccident)
   Result (CgetT)
EndTT
```

```
Cab :=
   Cin;
      (
         DaskC;
         (
            hide Caccident in
            Caccident; stop
            []
            CgetT; stop
         )
      ) (* L1 *)
```

Figure 47: Example of a loss

The STDL **`Loss`** segment becomes an alternate path with a hidden gate (here *Caccident*) followed by **`stop`**.

4.5.5 Tags

The following examples deal with the tag mechanism introduced in sections 4.1.3 and 4.1.4. In figure 46, the timethread *Traveler* is an adaptation of the timethread presented in figure 18. Here, we see that if a traveler does not phone a dispatcher, the tag *M* becomes equal to *Bus* and then the traveler cannot take a cab (*Tcab*) to get to the airport.

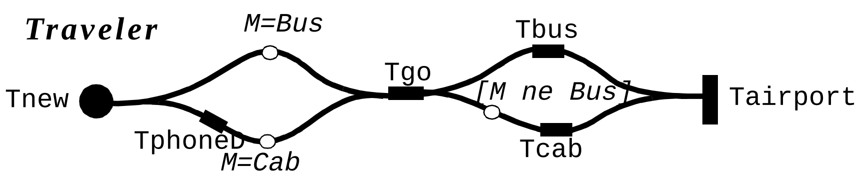


Figure 48: Example of tags

The STDL **`Tag`** and **`Guard`** constructs are used to express assignations to tags and guarded alternatives (fig. 46).

```
Timethread Traveler is

   Trigger (Tnew)
   Choice
         Tag (M = Bus)
      Or
         Action (TphoneD)
         Tag (M = Cab)
   EndChoice

   Action (Tgo)

   Choice
         Action (Tbus)
      Or
         Guard (M ne Bus)
         Action (Tcab)
   EndChoice


   Result (Tairport)

EndTT
```

```
Traveler :=
   Tnew;
   (
      hide SyncOr1 in
      (
         (let M: Tag = Bus in SyncOr1 ! M; stop)
         []
         TphoneD;
         (let M: Tag = Cab in SyncOr1 ! M; stop)
      )
      |[SyncOr1]|
      SyncOr1 ? M:Tag;
      Tgo;
      (
         hide SyncOr2 in
         (
            (Tbus; SyncOr2 ! M; stop)
            []
            ([M ne Bus]-> Tcab; SyncOr2!M; stop)
         )
         |[SyncOr2]|
         SyncOr2 ? M:Tag;
         Tairport; stop
      )
   )
```

Figure 49: Example of tags in STDL and LOTOS

Assignations of values to tags are done via the LOTOS **`let`** operator. Mechanisms to transfer tags on synchronization points (*SyncOr1* and *SyncOr2*) are essential to maintain the availability of tags all along a timethread path. We also interpret STDL guards as LOTOS guards without any transformation. During the mapping process, the type *Tag* is created:

```
type Tag is Boolean, NaturalNumber
sorts Tag
opns Bus,
     Cab  :  -> Tag
     N : Tag -> Nat          (* Tag-to-natural mapping operator *)
     _eq_,
     _ne_ : Tag, Tag -> Bool (* Tag equivalence operators *)
eqns forall x, y: Tag
    ofsort Nat
    N(Bus) = 0;              (* Bus is mapped onto 0 *)
    N(Cab) = Succ(N(Bus));   (* Cab is mapped onto 1 *)
    ofsort Bool
    x eq y = N(x) eq N(y);
    x ne y = not(x eq y);
endtype
```

This abstract data type provides constructors (*Bus* and *Cab*) and natural-based equivalence operations (*eq* and *ne*). Boolean operations are also supported (*not, and*, *or*, *xor, implies*, and *iff*).

Tag flow is interpreted in the same way. In figure 46, we modify the previous example in order to get two separate timethreads, *Traveler* and *Transportation*. The tag *M* is passed to the second timethread at the synchronization point *Tgo* and then becomes the tag *C*.

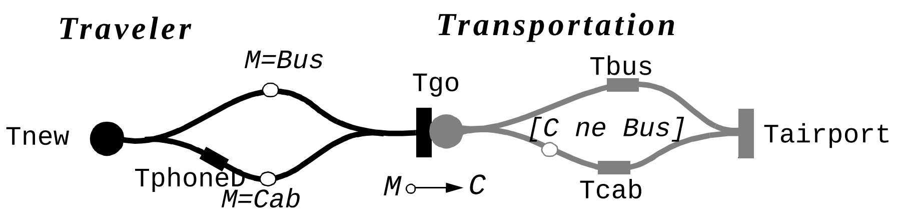


Figure 50: Example of tag flow

The STDL descriptions and LOTOS mapping are very similar to the previous example, except that there are two timethreads now synchronizing on gate *Tgo*. This mechanism is simple and yet powerful.

```
Timethread Traveler is

   Trigger (Tnew)
   Choice
         Tag (M = Bus)
      Or
         Action (TphoneD)
         Tag (M = Cab)
   EndChoice
   Result (Tgo ! M)

EndTT
```

```
Traveler :=
   Tnew;
   (
      hide SyncOr1 in
      (
         (let M: Tag = Bus in SyncOr1 ! M; stop)
         []
         TphoneD;
         (let M: Tag = Cab in SyncOr1 ! M; stop)
      )
      |[SyncOr1]|
      SyncOr1 ? M:Tag;
      Tgo ! M; stop
   )
```

```
Timethread Transportation is

   Trigger (Tgo ? C:Tag)
   Choice
      (
         Action (Tbus)
      Or
         Guard (C ne Bus)
         Action (Tcab)
      )
   Result (Tairport)

EndTT
```

```
Transportation :=
   Tgo ? C:Tag;
   (
      hide SyncOr2 in
      (
         (Tbus; SyncOr2 ! C; stop)
         []
         ([C ne Bus] -> Tcab; SyncOr2 ! C; stop)
      )
      |[SyncOr2]|
      SyncOr2 ? C:Tag;
      Tairport; stop
   )
```

Figure 51: Example of tag flow: STDL and LOTOS

4.6 Mappings

The question that could be raised at this point is whether the process of translating STDL constructs into LOTOS could be defined precisely. Since showing this completely would require considerable space, we will limit ourselves to providing a simple mapping table (fig. 52) which should show that this can be done, without entering into details. Only the rewrite rules that affect the LOTOS code are enumerated. These mappings are part of the mapping method that is to be implemented by a compiler.

How to Read this Table

- The left side shows STDL constructs and their context while the right side presents the corresponding LOTOS code generated.
- Characters in **`bold`** indicate code input from the STDL source or code output to the LOTOS destination.
- Characters in *`italic`* represent grammar rules that are copied as is from STDL input to LOTOS output. No special mappings are necessary for these rules.
- Rules between brackets (`<Rule>`) refer to the grammar rules. `<ROP>` is however not a grammar rule, but it refers to the Rest Of Path, which usually is `{<Seg>} <Result>`.
- These mappings generate level 1 specifications without recursion. We present and identify other levels (level 1 with recursion and level 3) for pertinent rules only. Level 2 is not considered in this table.
- Optional rules are between `[` and `]`, and list of rules are between `{` and `}`.
- Comments are between `(*` and `*)`.
- Lists of LOTOS gates (such as `GateList`) are computed from the list of activities found in a timethread description. We do not tell how they are computed here.

Example

As a short example, we can take a look at the `<Async>` rule:

(* Rule <Async> *)
Async (*EventId* [*SendTagValues*]**)** <ROP>
	(
	   EventId [*SendTagValues*]**; stop**
	   |||
	   <ROP>
	)

If part of the STDL code (with rules) is

Async (DaskC ! Tag1 ! Tag2) <Seg$_1$> <Seg$_2$> <Result>

Then the LOTOS code generated is:

```
(
   DaskC ! Tag1 ! Tag2; stop
   |||
   <Seg1> <Seg2> <Result>
)
```

An then `<Seg`$_1$`>` `<Seg`$_2$`>` and `<Result>` are replaced with their own corresponding LOTOS code.

Mapping Table

Input STDL Code	Output LOTOS Code

(* Rule <Timethread> *)
Timethread *TTid* **Is**
   <StubOrTT>
EndTT
	process *TTid* **[**GateList**] : noexit :=**
	   <StubOrTT>
	endproc

(* Rule <Stub> *) **Stub** <Trigger> <Result> **EndStub**	<Trigger> **(** <Result> **)** or, at a level 3: <Trigger> **(** <Result> **|||** *TTid* **[**GateList**]** (* refers to <Timethread> *) **)**

(* Rule <Aborted> *) **AbortedOn (***EventId***)** [<Internals>] <Trigger> <FirstPath>	**hide** *EventId* **in** **(** [<Internals>] <Trigger> <FirstPath> **)** **[>** *EventId***; stop** or, at level 1 with recursion : **hide** *EventId* **in** **(** [<Internals>] <Trigger> <FirstPath> **)** **[>** *EventId***;** *TTId* **[**GateList**]** (* refers to <Timethread> *)

(* Rule <Constrained> *)
Constrained
[<Internals>] <Trigger> {<Seg>} <Result>
	At level 1, it has no impact. At level 3 we find:
	
	hide SyncCS in
	   [<Internals>]
	   WP_CS [GateList**]**
	   |[SyncCS]|
	   *TTId***Sub [**GateList**]**
	
	where
	
	   process WP_CS [GateList**] : noexit :=**
	      <Trigger>**;**
	      (SyncCS; stop
	         |||
	         WP_CS [GateList] **)**
	   endproc
	
	   process *TTId***Sub [**GateList**] : noexit :=**
	      SyncCS;
	      {<Seg>}
	      *TTId***Sub [**GateList**]**
	   endproc

(* Rule <Internal> *) **Internal** *Identifier* {**,** *Identifier*}	**hide** *Identifier* {**,** *Identifier*} **in**
(* Rule <Trigger> *) **Trigger (***TriggerId* [*RecTagValues*]**)**	*TriggerId* [*RecTagValues*]**;**
(* Rule <FirstPath> *) <FirstPath>	**(** {<Seg>} <Result> **)** or, at level 3 : **(** {<Seg>} <Result> **|||** *TTid* **[**GateList**]** **)**
(* Rule <Result> *) **Result (***ResultId* [*SendTagValues*]**)**	*ResultId* [*SendTagValues*]**; stop** or, at level 1 with recursion : *ResultId* [*SendTagValues*]**;** *TTid* **[**GateList**]**

(* Rule <Delayed> used with Wait *)
(* Similar approach for Trigger *)
Wait Delayed (*EventId* [*RecTagValues*]**)** <ROP>
	(
	   hide Delay in
	      Delay;
	      EventId [*RecTagValues*];
	      <ROP>
	)

(* Rule <Time> used with Wait *)
(* Similar approach for Trigger *)
Wait Time (*EventId* [*RecTagValues*]**)** <ROP>
	(
	   hide TimeOut, SyncTO in
	   (
	      TimeOut;
	      SyncTO! TOut; stop
	      []
	      EventId [*RecTagValues*];
	      SyncTO! OK; stop
	   )
	   |[SyncTO]|
	   SyncTO ?ResultTO:Tag;
	   <ROP>
	)

(* Rule <Abort> *) **Abort (***EventId***)**	*EventId*;
(* Rule <Action> *) **Action (***ActionId***)**	*ActionId*;

(* Rule <Async> *)
Async (*EventId* [*SendTagValues*]**)** <ROP>
	(
	   EventId [*SendTagValues*]; **stop**
	   |||
	   <ROP>
	)

(* Rule <Sync> *) **Sync (***EventId* [*RecTagValues*]**)**	*EventId* [*RecTagValues*];

(* Rule <Choice> *)
Choice
   [<Guard$_{One}$>] {<Seg$_{One}$>}
   { **Or** [<Guard$_{Next}$>] {<Seg$_{Next}$>} }*
EndChoice
<ROP>
	(
	 hide SyncOR in
	   ([<Guard$_{One}$>] {<Seg$_{One}$>} **SyncOR; stop**
	    {[] [<Guard$_{Next}$>] {<Seg$_{Next}$>} **SyncOR;stop** }*
	   )
	   |[SyncOR]|
	   SyncOR;
	   <ROP>
	)

(* Rule <OrFork> *)
OrFork
   [<Guard$_{Cont}$>] **Continue**
   { **Or** [<Guard$_{Next}$>] <Path> }*
EndOrFork
<ROP>
	(
	   [<Guard$_{Cont}$>] <ROP>
	   { **[]** [<Guard$_{Next}$>] <Path> }*
	)

(* Rule <Path> *) **Path** {<Seg>} <Result> **EndPath**	{<Seg>} <Result>

(* Rule <Loop> *)
Loop
   <CompSymb>
      [Guard$_{Comp}$] {<Seg$_{Comp}$>}
   <OptSymb>
      [Guard$_{Opt}$] {<Seg$_{Opt}$>}
EndLoop
<ROP>
	Loop [GateList**]**
	... (* Closing parenthesis, if any *)
	where
	   process Loop [GateList**] : noexit :=**
	      {<Seg$_{Comp}$>}
	      (
	         [Guard$_{Opt}$] {<Seg$_{Opt}$>}
	         Loop [GateList**]**
	         []
	         [<Guard$_{Comp}$>] <ROP>
	      )
	   endproc

(* Rule <Loss> *)
Loss ([<Guard>] *LossId***)** <ROP>
	(
	   hide *LossId* **in**
	      [Guard] *LossId*; **stop**
	      []
	      <ROP>
	)

(* Rule <Par> *)
Par
   {<Seg$_{One}$>}
   { **And** {<Seg$_{Next}$>} }*
EndPar
<ROP>
	(
	 hide SyncAND in
	   ({<Seg$_{One}$>} **SyncAND; stop**
	     {**|[SyncAnd]|** {<Seg$_{Next}$>} **SyncAND; stop** }*
	   )
	   |[SyncAND]|
	   SyncAND;
	   <ROP>
	)

(* Rule <AndFork> *)
AndFork
   <Path$_{One}$>
   { **And** <Path$_{Next}$> }
EndAndFork
<ROP>
	(
	   <Path$_{One}$>
	   { **|||** <Path$_{Next}$> }
	   |||
	   <ROP>
	)

(* Rule <SegStub> *) **SegStub (***SegStubId***)**	*SegStubId*;
(* Rule <Wait> *) **Wait (***EventId* [*RecTagValues*]**)**	*EventId* [*RecTagValues*];
(* Rule <Tag> *) **Tag (***TagId* **=** *ValueId***)** <ROP>	**(let** *TagId* **: Tag =** *ValueId* **in** <ROP>**)**
(* Rule <Guard> *) **Guard (***GuardExpr***)** <ROP>	**([***GuardExpr***] ->** <ROP>**)**

Figure 52: Mappings from STDL to LOTOS

CHAPTER 5

Elements of a Life-Cycle Methodology

5.1 Introduction

The previous chapters introduced the LOTOS interpretation method for timethreads. We now relate this method to real-time and distributed (RTD) systems design. To do so, we first present a short overview of a timethread-oriented life-cycle methodology (§5.1.1). Then, different techniques related to this methodology are discussed. Section 5.2 presents the complete mapping procedure of a timethread map onto LOTOS. We use the *Traveler System* introduced in section 3.3 to generate two specifications (level 1 and level 3). We shortly discuss a few transformation techniques in section 5.3. Finally, we present validation techniques in section 5.4. Our transformation and validation techniques are based on LOTOS techniques and tools for simulation, testing, and verification.

5.1.1 Overview of a Life-Cycle Methodology

We can design RTD systems with timethreads in many different ways. Figure 53 presents an instance of a partial life-cycle (from requirements to the implementation) oriented towards timethreads. We used a timethread to show the sequence of activities in this approach.

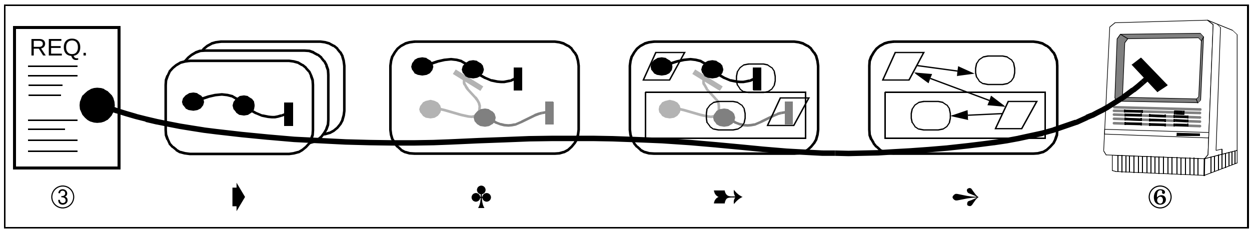


Figure 53: Elements of a timethread-oriented life-cycle

In [BuC 93], the authors discussed the timethread-centered design of RTD systems. We relate it here to the previous figure:

③ ***Requirements***: They are usually expressed in natural language (English). They describe the functionalities of a system to be built.

➧ ***Requirements Capture***: Different scenarios are generated in order to discover the functionalities to be expressed with timethreads.

♣ ***Composition***: The different timethreads are composed together to form a timethread map which presents end-to-end causality paths within the system (the big picture). This process usually involves several timethread transformations. Different techniques to get a formal specification, to support transformations, and to validate the resulting map against the requirements (or previous maps) can be used.

➼ ***Components Discovery***: Components are associated to timethread activities. They can be given in the requirements or discovered along the design process. The output is called a *timethread-role map*. A role notation is proposed in [BuC 93], where architectural components are of three types: *carriers*, *workers*, and *teams*.

➾ ***Collaboration***: From the constraints expressed in the causality flow of the timethread-role map, we derive a more detailed *control flow* between components. In the resulting *collaboration graph*, containing architectural components and control paths drawn as arrows, timethreads are no longer needed.

⑥ ***Implementation***: This is the final, working system. Of course, many intermediate steps, such as the components design and the overall validation, are here left absent between the collaboration (➩) and the implementation (⑥). This is because timethreads are less useful when we get closer to the implementation, so we concentrate on the early steps of the life-cycle.

This life-cycle is one among the many possible ones that can use timethreads for RTD system design. We adopted this one because it concentrates first on the functionalities and the end-to-end causality paths, and then on the architectural components. In this way, we enhance the usefulness of timethreads.

In this thesis, we do not attack architectural problems. This is a complex topic that will have to be discussed in other theses. Therefore, the mapping, transformation and validation techniques of the current chapter are related to the phases that concern timethreads only (➧ and ♣).

5.1.2 Limitations of the Proposed Techniques

The mapping, transformation, and validation techniques we propose in the current chapter are not unique nor perfect, although they represent a few ideas that could be developed much further. Most of them are not formally defined and have not been thoroughly tested yet. Proving in any way that such methods work properly and efficiently is in fact a very complex task for anyone. Because this is only preliminary work, we present these techniques using simple examples related to the *Traveler System*.

5.2 Mapping Techniques

Section 3.1 presented our solution to the timethread-map-to-LOTOS mapping. The *Traveler System* can help us illustrate this mapping. The following four subsections use four methods (map decomposition, LAEG, timethread mapping, and composition) for the generation of a LOTOS specification corresponding to the *Traveler System* timethread map. Note that we do not develop further the timethread mapping method introduced in chapter 4. In order to do so, we would have to write a compiler, and this still represent a large amount of work.

5.2.1 Map Decomposition

The map decomposition method generates a LARG representing timethreads interactions, and the SDTL representation of the timethreads included in the map. We also decide here which activities are to be hidden from a LOTOS viewpoint.

Hiding

The hiding of activities defines the interface of systems and processes. This is usually a designer's decision, mostly related to a topology of components in an architecture. However, we propose here two simple hiding rules for timethread maps which do not have any commitment to a specific architecture yet. We suppose that a system is represented as a *box* on which we superimpose timethreads:

- Activities within the box (system's boundaries) are globally hidden.
- Actions along a timethread path are locally hidden. We consider them as internal.

In the *Traveler System* (fig. 15), we consider that all activities are internal (globally hidden), except for *Tnew* and *Tdest* which represent the interface of the system. Actions such as *Cgarage*, *Dready*, and *Tairport* are locally hidden within the LOTOS processes corresponding to their respective timethreads. Note that *TCride* and *TPflight*, although interpreted as `Action` in STDL, are considered as events because timethreads have to synchronize on these gates. Therefore, we cannot hide them locally; these actions are part of the interface of their respective timethreads.

These rules could be modified in a tool. LOTOS offers much flexibility w.r.t. gate hiding, and the LOTOS interpretation method for timethreads is still valid w.r.t. whichever choices are made.

LARG

The first step of the map decomposition method is the generation of a LARG representing the topology of interacting timethreads. Figure 54 shows the LARG of the traveler system. There are four processes corresponding to the four timethreads in the map. These processes interact on different events which are included in the gate sets. We observe that the interface of the system *Traveler_Example* is composed of two events: *Tnew* and *Tdest*. All other activities are hidden globally. Due to a lack of space, we do not show all activities in the LARG, but we use italicized label sets that enumerate them below the diagram.

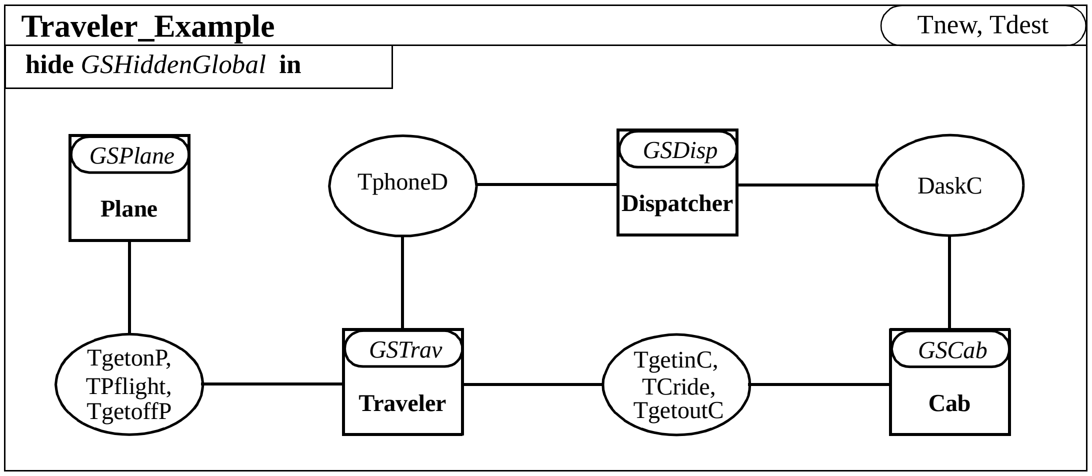


Figure 54: LARG of the Traveler System

The italicized label sets correspond to the following gates:

- *GSHiddenGlobal* is `TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPFlight, TgetoffP, Pready, Phangar, Din, DaskC, Dout, Cin, Cout`
- *GSPlane* is `Pready, TgetonP, TPFlight, TgetoffP, Phangar`
- *GSTrav* is `Tnew, TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPFlight, TgetoffP, Tdest`
- *GSDisp* is `Din, TphoneD, DaskC, Dout`
- *GSCab* is `Cin, DaskC, TgetinC, TCride, TgetoutC, Cout`

The actions locally hidden within timethreads (not shown in the diagram) are:

- *Plane*: No action is hidden.
- *Traveler*: `Tairport`
- *Dispatcher*: `DlookforC, Dfillstats, Dready`
- *Cab*: `CgoD, Cgarage`

STDL

The second step consists in describing the paths of individual timethreads in STDL. Four descriptions are needed, one for each timethread (fig. 55).

```
Timethread Traveler is
   Internal Tairport
   Trigger (Tnew)
   Async (TphoneD)
   Sync (TgetinC)
   Action (TCride)
   Sync (TgetoutC)
   Action (Tairport)
   Sync (TgetonP)
   Action (TPflight)
   Sync (TgetoffP)
   Result (Tdest)
EndTT
```

```
Timethread Dispatcher is
   Constrained
   Internal DlookforC, Dfillstats, Dready
   Trigger (Din)
   Loop
      Compulsory
         Sync (TphoneD)
         Action (DlookforC)
         Async (DaskC)
         Action (Dfillstats)
      Optional
         Action (Dready)
   EndLoop
   Result (Dout)
EndTT
```

```
Timethread Cab is
   Constrained
   Internal CgoD, Cgarage
   Trigger (Cin)
   Loop
      Compulsory
         Sync (DaskC)
         Sync (TgetinC)
         Action (TCride)
         Sync (TgetoutC)
      Optional
         Action (CgoD)
   EndLoop
   Action (Cgarage)
   Result (Cout)
EndTT
```

```
Timethread Plane is
   Trigger (Pready)
   Sync (TgetonP)
   Action (TPflight)
   Sync (TgetoffP)
   Result (Phangar)
EndTT
```

Figure 55: STDL descriptions of timethreads in the Traveler System

These descriptions follow the rules and examples presented in the previous chapter.

5.2.2 Application of LAEG Method

In order to generate LOTOS structural expressions from LARGs, we have to analyze the LARG and transform it into a binary grouped one. A binary grouped LARG allows a direct mapping onto LOTOS.

LARG Analysis

By applying the LARG analysis defined in [Bor 93], we find no structural ambiguity (see §2.3.2). Therefore, we can use the Grouping algorithm to generate a binary grouped LARG equivalent to the LARG of figure 54.

Many different groupings can result from this algorithm. Design decisions such as performance and location of components and/or processes should indicate which grouping is the best. However, no such metrics have been defined yet. Since we think that we should not be concerned with such problems at a timethread level, any grouping that preserves the semantics or the original LARG is valid for execution and validation. Figure 56 shows one possible grouping.

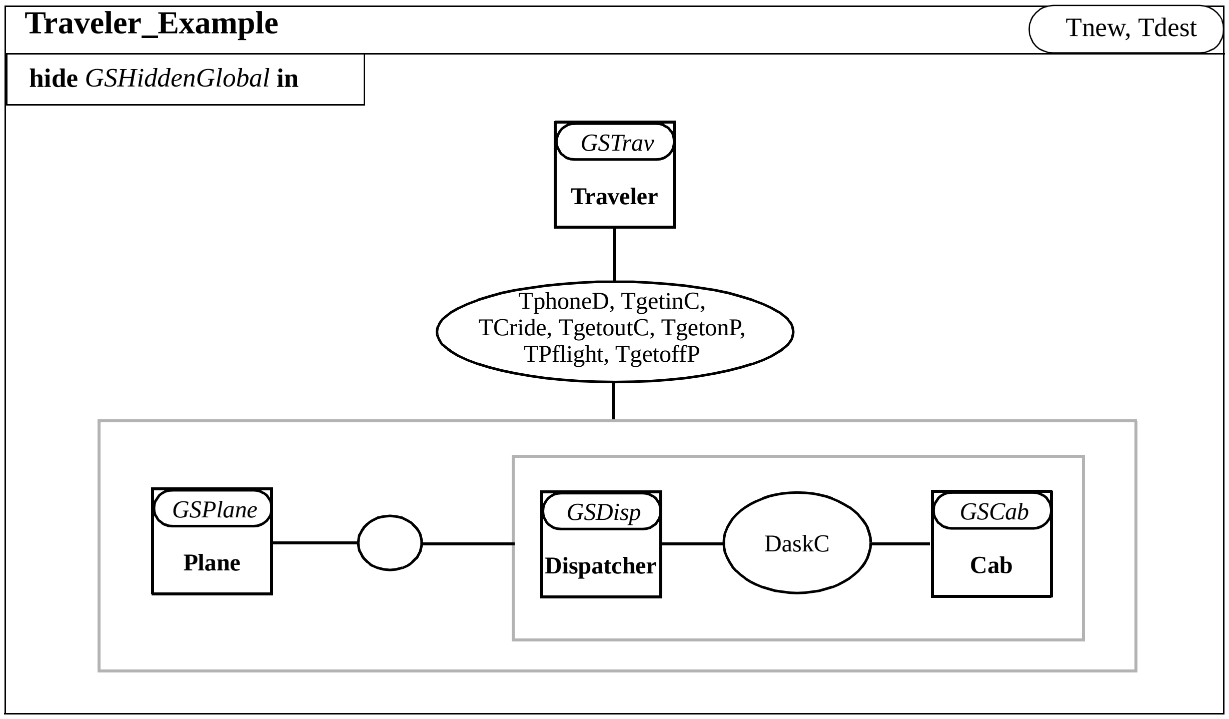


Figure 56: Binary grouped LARG of the Traveler System

In this equivalent LARG, hidden gates (global and local to timethreads) and interfaces have not been modified in the processes and the original semantics is preserved. Note that an empty gate set, such as the one linked to *Plane*, means that the processes involved are interleaving.

Generation LOTOS Structural Expression

From the binary grouped LARG, we can derive the structural section, called *behaviour* section in LOTOS, of the LOTOS specification (see [Bor 93] for more details). Note that a grouping is represented in LOTOS as parenthesis. Specification 3 presents the interfaces of the system and processes (with hidden activities), and the interacting processes. Line numbers are not part of specifications; they are used for referencing the code.

```
1    (* Traveler_Example; Daniel Amyot, March 29, 1994 *)
2    (* Level 1 specification of the Traveler system *)
3
4    specification Traveler_Example[Tnew  (* New traveler wants to travel *),
5                                   Tdest (* Traveler arrives to destination *) ] : noexit
6
7    behaviour (* Structure obtained from the LARG *)
8
9    hide  (* hidden interactions GSHiddenGlobal *)
10       TphoneD,        (* Traveler phones Dispatcher for a cab *)
11       TgetinC,        (* Traveler gets in the cab *)
12       TCride,         (* Traveler and cab ride *)
13       TgetoutC,       (* Traveler gets out the cab *)
14       TgetonP,        (* Traveler gets on the plane *)
15       TPflight,       (* Traveler and plane flight *)
16       TgetoffP,       (* Traveler gets off the plane *)
17       Din,            (* Dispatcher is in the office *)
18       DaskC,          (* Dispatcher asks for a cab *)
19       Dout,           (* Dispatcher is not in the office *)
20       Cin,            (* Taxi driver in the cab *)
21       Cout,           (* Taxi driver not in the cab *)
22       Pready,         (* Plane is ready *)
23       Phangar         (* Plane goes to the hangar *)
24
25   in
26
27       Traveler[Tnew, TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPflight, TgetoffP, Tdest]
28       |[TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPflight, TgetoffP]|
29       (
30           Plane[Pready, TgetonP, TPflight, TgetoffP, Phangar]
31           |||
32           (
33               Dispatcher[Din, TphoneD, DaskC, Dout]
34               |[DaskC]|
35               Cab[Cin, DaskC, TgetinC, TCride, TgetoutC, Cout]
36           )
37       )
38
39   where
40
41   (*  Local hidden actions:              *)
42   (*  --------------------               *)
43   (*  Traveler: Tairport                         *)
44   (*  Plane:                                     *)
45   (*  Dispatcher: DlookforC, Dfillstats, Dready  *)
46   (*  Cab: CgoD, Cgarage                         *)
47
48   (* Process definitions have to be included here *)
49   endspec (* Specification Traveler_Example *)
```

Specification 3: Structure obtained from figure 56

For the specification to be complete, process definitions generated from STDL descriptions have to be included between lines 47 and 49.

5.2.3 Generation of Behaviour Expressions

To generate the missing processes in specification 3, we generate LOTOS behaviour expressions from the STDL descriptions of figure 55. We use here the mapping method introduced in section 4.3 to generate four processes corresponding to the four timethreads in the *Traveler System* (specifications 4 to 7).

It must be understood that we dot not, at present time, have a complete algorithm to translate SDTL descriptions into LOTOS processes. Such an algorithm is the object of further research. However, by using the principles developed in chapter 4, a manual and intuitive translation process (similar to the process of obtaining code from a flowchart) is possible. We also believe that a compiler could automate this translation process. For instance, such compiler would have to:

- Implement lexical and semantical analysis.
- Manage gate parameters.
- Manage Abstract Data Types (for *tags*): type definition, message passing, tags availability, consistency...
- Generate the structure from the interaction part of the *LARG* description. This was already introduced in the *LAEG* method.
- Generate LOTOS processes corresponding to single timethreads.
- Manage unique names for additional internal synchronization gates.
- Etc.

By manually translating SDTL to LOTOS, we get the four following processes:

Timethread Traveler

This process is straightforward to generate. We simply have a sequence of activities. The only difficulty resides in the asynchronous event *TphoneD*.

```
50  (* Timethread Traveler *)
51      process Traveler[Tnew, TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPflight, TgetoffP,
52                         Tdest] : noexit :=
        hide Tairport in  (* hidden action *)
        Tnew;
        (
            TphoneD; stop  (* in-passing interaction *)
            |||
            (
                TgetinC;   (* rest of the path *)
                TCride;
                TgetoutC;
                Tairport;
                TgetonP;
                TPflight;
                TgetoffP;
                Tdest; stop
            )
        )
    endproc (* Traveler *)

(*-------------------------------------------------------*)
```

Specification 4: Process Traveler

Timethread Dispatcher

A sub-process *DispatcherLoop* is needed here to simulate the loop part of the timethread. Also, we have the asynchronous event *DaskC* which complicates the process generation.

```
(* Timethread Dispatcher *)
    process Dispatcher[Din, TphoneD, DaskC, Dout] : noexit :=
        (* hidden actions *)
        hide
            DlookforC,   (* Dispatcher looks for a cab *)
            Dfillstats,  (* Dispatcher fills statistics *)
            Dready       (* Dispatcher is ready for next traveler *)
        in
        Din; DispatcherLoop[TphoneD, DlookforC, DaskC, Dfillstats, Dready, Dout]
        where

        process DispatcherLoop[TphoneD, DlookforC, DaskC, Dfillstats, Dready, Dout]: noexit:=
            (* Compulsory segment *)
            TphoneD;
            DlookforC;
            (
                DaskC; stop  (* in-passing interaction *)
                |||
                Dfillstats;
                (
                    (* Optional segment *)
                    Dready; DispatcherLoop[TphoneD,DlookforC, DaskC,Dfillstats, Dready, Dout]
                    []
                    (* Exit Loop *)
                    Dout; stop
                )
            )
        endproc (* DispatcherLoop *)
    endproc (* Dispatcher *)

(*-------------------------------------------------------*)
```

Specification 5: Process Dispatcher

Timethread Cab

This process needs a sub-process *CabLoop* to manage the loop part of its corresponding timethread.

```
(* Timethread Cab *)
    process Cab[Cin, DaskC, TgetinC, TCride, TgetoutC, Cout] : noexit :=
        (* hidden actions *)
        hide
            CgoD,    (* Cab goes to wait the dispatcher *)
            Cgarage  (* Cab goes to the garage *)
        in
        Cin; CabLoop[DaskC, TgetinC, TCride, TgetoutC, CgoD, Cgarage, Cout]
        where

        process CabLoop[DaskC, TgetinC, TCride, TgetoutC, CgoD, Cgarage, Cout] : noexit :=
            (* Compulsory segment *)
            DaskC;
            TgetinC;
            TCride;
            TgetoutC;
            (
                (* Optional segment *)
                CgoD; CabLoop[DaskC, TgetinC, TCride, TgetoutC, CgoD, Cgarage, Cout]
                []
                (* Exit Loop *)
                Cgarage;
                Cout; stop
            )
        endproc (* CabLoop *)
    endproc (* Cab *)

(*-------------------------------------------------------*)
```

Specification 6: Process Cab

Timethread Plane

This is the easiest process of the four. We simply have a sequence of activities which is interpreted in the following way:

```
(* Timethread Plane *)
    process Plane[Pready, TgetonP, TPflight, TgetoffP, Phangar] : noexit :=
        (* no hidden action in the timethread *)
        Pready;
        (
            TgetonP;
            TPflight;
            TgetoffP;
            Phangar; stop
        )
    endproc (* Plane *)
```

Specification 7: Process Plane

5.2.4 Composition of the Complete Specification

The complete specification is obtained simply by adding the process definitions to the end of the LOTOS structure generated from the LARG. If a timethread map contains tags, an abstract data type *Tag* must also be added. This ADT could be automatically generated from the timethreads STDL descriptions.

The specification developed here is at a level 1 (§4.1.2). Other levels can be used. The specification in appendix B (*Traveler_Level3*) is the level 3 interpretation of the *Traveler System*. The modifications brought to the current level 1 specification to transform it to a level 3 specification are *italicized*. The transformation of processes *Traveler* and *Plane* are straightforward, but processes *Dispatcher* and *Cab* need to have a more complex mechanism to manage constrained-start timethread in a recursive environment. The structure of this specification is kept unchanged.

5.3 Transformation Techniques

The purpose of this section is to give a short overview of several simple timethread transformations (see §2.1.4). The categorization of these transformations, their impact, and their enumeration is still an ongoing research topic (see [BoL 94]). Nevertheless, we introduce here a few transformation techniques and their impact on SDTL and LARG descriptions, and on the resulting LOTOS specification. We also relate these transformations to LOTOS equivalence and extension relations. Two short examples, which we reuse in chapter 6 (in the *Telepresence* system), will help us illustrate a few concepts, techniques, and new issues.

5.3.1 Equivalence Relations

The concept of transformation brings about some notions of relations (equivalence, extension, reduction, and conformance, as presented in [Led 91 and BSS 86]) because obviously timethreads that are one transformation of the other must be related in some sense.

Equivalence is by far the most interesting relation. LOTOS proposes many levels of equivalence between two specifications. These equivalence relations are found in many semantic models, including *Labeled Transition Systems* (LTS), the underlying model of LOTOS. Among the most popular relations related to LTS (see [BoB 87] and [CPT 92]), we find, from the weakest to the strongest relation:

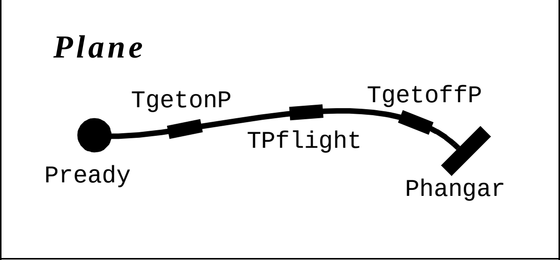
- Trace equivalence
- Testing equivalence
- Weak bisimulation equivalence
- Strong bisimulation equivalence
- Equality

Many other equivalence relations (other bisimulations, congruences) have been defined over the years, but it is not our goal to discuss them here. The point is that they all measure *behaviour equivalence*, i.e., whether or not two specifications behave in the same way according to some criteria.

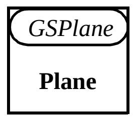
Timethreads do not fundamentally intend to express the behaviour of a system, but its causality paths. Hence, this difference leads to a concept that we could call *path equivalence*, which measures that two timethreads (or perhaps timethread maps) have equivalent causality paths. Path equivalence will remain informally defined here, as it is still an open issue. However, in our first example, we attempt to relate it to LOTOS equivalence relations.

5.3.2 First Example: Splitting a Sequence

One of the simplest timethreads is the sequence. We will use one derived from the *Traveler* example: the timethread *Plane* of figure 57. Its LARG, SDTL, and LOTOS descriptions are also presented in order to observe the complete impact of a transformation on this sequence.



LARG:



where *GSPlane* is:
`Pready, TgetonP, TPflight, TgetoffP, Phangar`

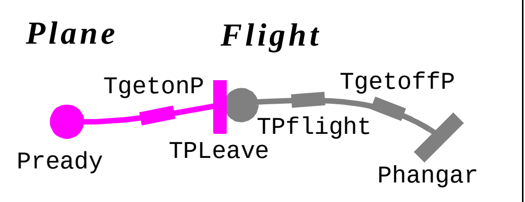
```
Timethread Plane is
   Trigger (Pready)
   Path
      Action (TgetonP)
      Action (TPflight)
      Action (TgetoffP)
      Result (Phangar)
   EndPath
EndTT
```

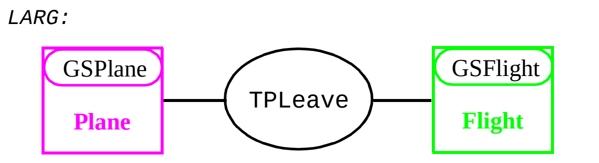
```
Plane :=
   Pready;
   (
      TgetonP;
      TPflight;
      TgetoffP;
      Phangar; stop
   ) (* L1 *)
```

Figure 57: Timethread Plane and its LARG, SDTL, and LOTOS descriptions

The transformation which interests us here is a timethread cutting followed by the composition of the two resulting parts. We would like the resulting map (composed of two interacting timethreads) to be equivalent in some way to the original sequence.

Assume we want to split our sequence in two interacting timethreads between the actions `TgetonP` and `TPflight`. We first cut the timethread `Plane` into a new `Plane` and a timethread `Flight`. Then, we compose them on a new <u>internal</u> action `TPLeave` to form the map shown in figure 58.





where *GSPlane* is: `Pready, TgetonP, TPLeave`
and *GSFlight* is: `TPLeave, TPflight, TgetoffP, Phangar`

```
Timethread Plane is
   Trigger (Pready)
   Path
      Action (TgetonP)
      Result (TPLeave)
   EndPath
EndTT

Timethread Flight is
   Trigger (TPLeave)
   Path
      Action (TPflight)
      Action (TgetoffP)
      Result (Phangar)
   EndPath
EndTT
```

```
hide TPLeave in
   Plane |[TPLeave]| Flight
where

Plane :=
   Pready;
   (
      TgetonP;
      TPLeave; stop
   ) (* L1 *)

Flight :=
   TPLeave;
   (
      TPflight;
      TgetoffP;
      Phangar; stop
   ) (* L1 *)
```

Figure 58: Transformation applied to the sequence of fig. 57

Such transformation has a profound impact on the SDTL and LARG descriptions. A new timethread is created, and so are a new process and a new interaction. The resulting LOTOS code is also different, although the resulting behaviour is strongly similar to the behaviour of the original specification.

With relation to the discussion on equivalence concepts (§5.3.1), the first question to ask here is whether or not this transformation preserves the original path, i.e., whether those two maps are *path equivalent* or not. Although we do not have a formal definition of path equivalence, the maps of figures 57 and 58 seem equivalent from a timethread viewpoint. A token originating at `Pready` in either map will follow the same path, lead to the same activities in the same order (except for the new activity `TPLeave` which is internal and therefore of no concern to us), and then give the same result without being able to go anywhere else.

Now, to relate this to LOTOS equivalence relations, we show the LTS of each map (level 1 without recursion) in figure 59:

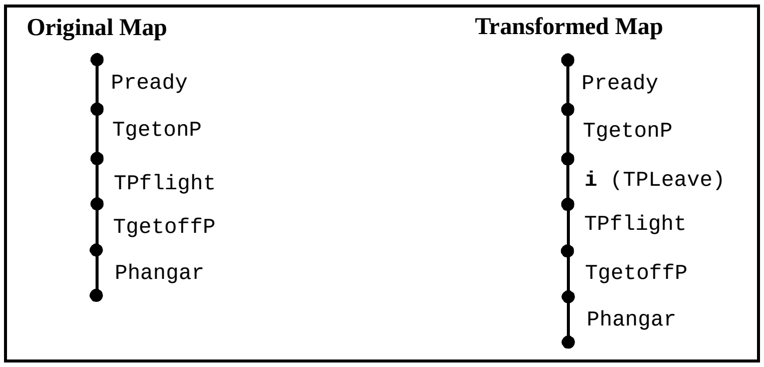


Figure 59: LTSs of maps of fig. 57 and fig. 58

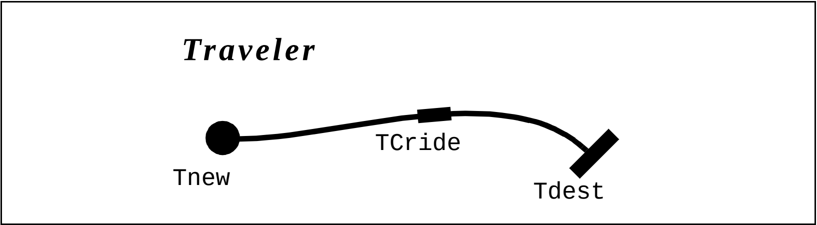
In this case we observe a weak bisimulation equivalence between the two maps, i.e., the two systems behave in the same way to any external observer. This is known as process splitting in [CPT 92] and [Lan 90]. However, if we have a level 1 specification <u>with recursion</u>, we cannot relate the resulting LTSs to any existing LOTOS equivalence relation. The reason is that in the second map, as soon as the `TPLeave` is reached in `Plane`, a new token can be placed on `Pready`, while the timethread `Plane` of the first map needs to wait until `Phangar` is reached before a new token is allowed to be placed on `Pready`. Therefore, an accumulation of tokens may occur on `TPflight` in the second map while this was impossible in the first map. The existing equivalence relations based on behaviour do not cover such concepts yet. Also, if we consider level 2 and 3 specifications, LTSs become more complex and other problems may arise.

This timethread transformation leads to a reorganization of SDTL and LARG descriptions, resulting in a path equivalent LOTOS specification. Of course, a new *LOTOS path equivalence* is to be defined according to our needs, but this is still a complex research issue. A tool could manage this type of technique by allowing transformations which are “assumed” to preserve path equivalence. It could also generate automatically the new SDTL and LARG descriptions.

5.3.3 Second Example: Extending a Sequence

The second example is concerned with other types of relation called *extensions*. These relations do not preserve equivalence between an original specification and a transformed one. They are related to functionality extension, as presented in [CPT 92]. In timethread terms, we say that we add alternative paths to a timethread.

Assume the short *Traveler* sequence presented in figure 60, where we also find the SDTL and LOTOS descriptions. The LARG is not given since the extension will not affect it.

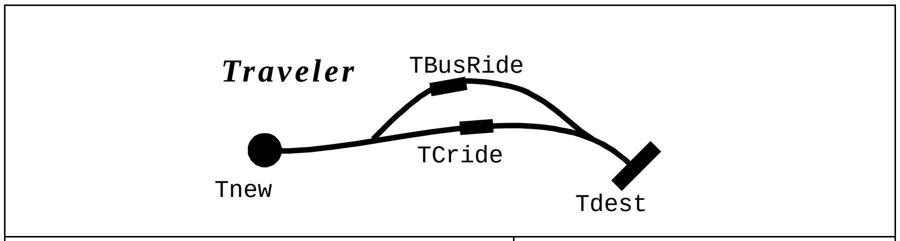


```
Timethread Traveler is
   Trigger (Tnew)
   Path
      Action (TCride)
      Result (Tdest)
   EndPath
EndTT
```

```
Traveler :=
   Tnew;
   (
      TCride;
      Tdest; stop
   ) (* L1 *)
```

Figure 60: Timethread Traveler and its SDTL and LOTOS descriptions

Now, suppose we want to extend the system by allowing the traveler to either take the cab, as usual, or the bus (`TBusRide`). To do so, we use a OR-Fork/OR-Join (SDTL **`Choice`** construct) where one branch has action `TCride` and the other branch has `TBusRide`. After the transformation, we get the timethread, SDTL description, and LOTOS process shown in figure 61.



```
Timethread Traveler is
   Trigger (Tnew)
   Path
      Choice
            Action (TBusRide)
         Or
            Action (TCride)
      EndChoice
      Result (Tdest)
   EndPath
EndTT
```

```
Traveler :=
   Tnew;
   (
      hide SyncOr in
      (
         TBusRide;
         SyncOr; stop
         []
         TCride;
         SyncOr; stop
      )
      |[SyncOr]|
      SyncOr;
      Tdest; stop
   ) (* L1 *)
```

Figure 61: Extension applied to the sequence of fig. 60

The resulting SDTL description sees its previous action `TCride` replaced with a choice between `TBusRide` and `TCride`. The gate `TBusRide` is also obviously added to the LARG interface (not shown here). Now, can we say that the resulting LOTOS process is related in any way to the original one? To answer this question, we can develop their LTSs (fig. 62).

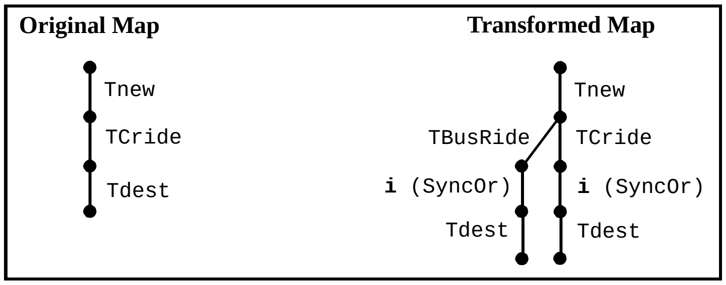


Figure 62: LTSs of processes of fig. 60 and fig. 61

At a level 1 (without recursion), such transformation leads a specification which *extends* the original specification. The LOTOS extension relation *ext* is formally defined in [BSS 87 and Led 91]. Intuitively, *S1* *ext* *S2* means that *S1* has more traces (due to additional paths) than *S2*, but it deadlocks less often in an environment whose traces are limited to those of *S2*. Therefore, this timethread transformation can be related to a LOTOS relation again.

Most timethread transformations could be related in one way or another to LOTOS conformance, equivalence, extension, and reduction relations. Although relations such as path equivalence have to be defined, we think that many LOTOS CPTs and relations could be adapted in order to define *Timethread Correctness Preserving Transformations* (TCPTs). These TCPTs could form the basis of transformations allowed in a timethread design tool. This is however an open issue we will not discuss further in this thesis.

5.4 Validation Techniques

Techniques to validate a design against given requirements are desirable. In the LOTOS world, such validation techniques have many aspects, and many tools exist to apply them. In the timethreads world, validation is still a research topic. What is to be validated and how this should be done is an open issue. However, we think we can apply some of the LOTOS validation techniques to a timethread map to get meaningful results.

Different aspects of a timethread map and its corresponding LOTOS specification can be analyzed or validated:

- We can *play* the design to see whether we captured all the pertinent information in the requirements. This can be done by executing the specification.
- We can discover concurrency, non-determinism, collision, and race problems that would have to be solved at a later stage when the design gets closer to the architecture.
- The executability of LOTOS can help us understand problems related to the ordering of events and to interactions between timethreads.
- Properties such as the absence of unwanted deadlocks can be verified in a system's design.
- We can verify whether or not a refined design conforms to a previous design, in order not to introduce or remove functionalities inadvertently, or to check consistency between successive designs.
- Test cases derived from previous designs can be used to validate new designs.

This enumeration is not exhaustive. Other validation aspects exist, but we consider the ones mentioned to be among the most important ones for LOTOS. In a multi-formalism approach, other validation aspects, such as performance requirements, could be added. However, we would need other formalisms suitable for these new aspects (e.g., Petri nets) and corresponding interpretation methods.

We consider that LOTOS specifications generated from timethread maps are not full behaviour specifications. In a complete *Traveler System*, it is unlikely that a cab or a plane will accept one and only one traveler at a time. This illustrates a major difference between a *path specification* and a *behaviour specification*. We do not consider every detail at the timethread abstraction level. Behaviour issues are usually solved later at the architecture level (not discussed in this thesis), and then the validation strategies are changed accordingly.

A timethread is a scenario or a path description. It is not intended to be a simulation or a prediction model. However, it is still useful to simulate the path behaviour described by a timethread. We can test, for example, that a sequence of causes *a; b; c;* will cause the effect *d;* and the rest of the timethread path.

In order to validate a timethread map, LOTOS provides us interesting facilities such as:

- The capacity to focus on single timethreads or on a topology of interacting timethreads,
- The use of different levels of specifications,
- The availability of three major validation techniques: simulation, testing and verification.

We must recall here that what we really validate is the LOTOS specification, which is a projection of a timethread map. Because of the semantic gap between timethreads and LOTOS, we cannot validate every aspect of a timethread map.

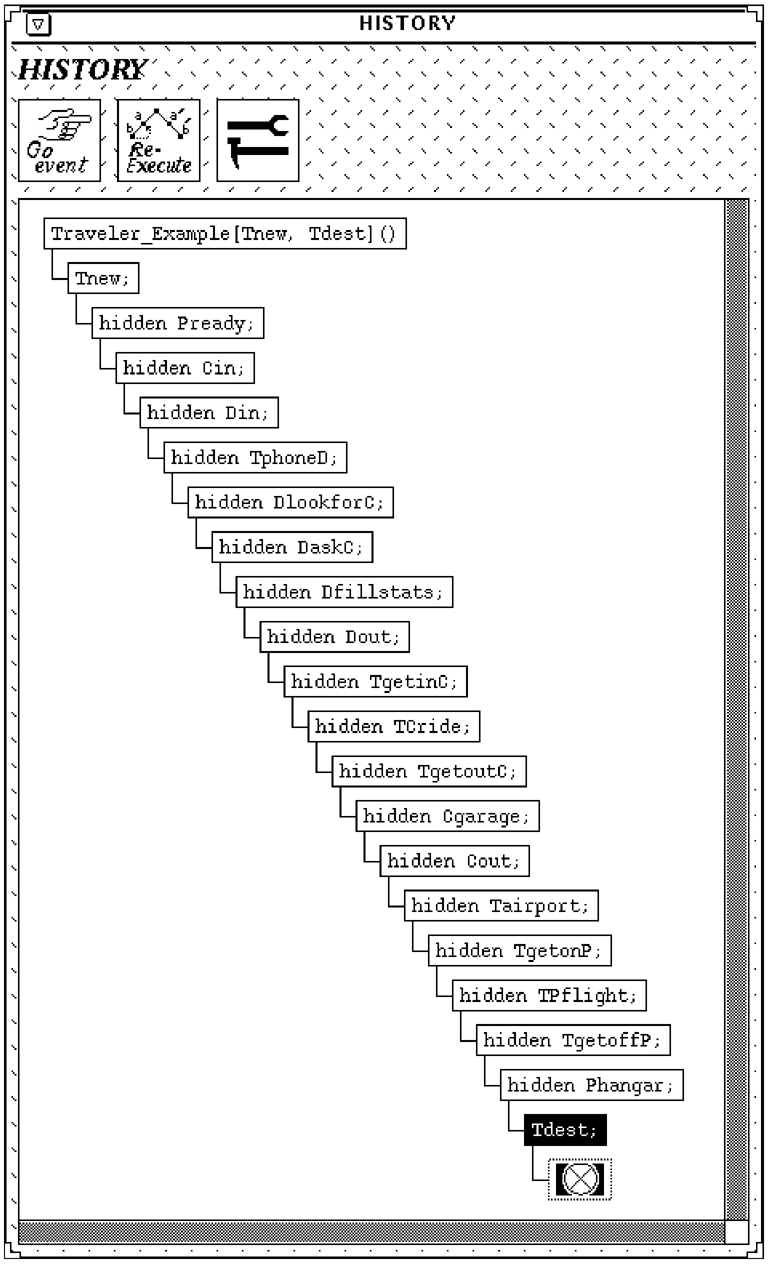
Diagnostics are here based on *names* in the map. We preserve the identifiers of timethreads (processes), activities (gates), tags (value identifiers), and values (ADT) in the mapping procedure. Therefore, a problem found using LOTOS validation tools can be related directly to its corresponding problem in the timethread map, simply because the names correspond.

5.4.1 Interactive Simulation

We consider the specification *Traveler_Example* (spec. 3) to be useful mostly because of its executability. Simulating such a LOTOS specification helps the designer to ensure that the timethread map corresponds to the functionality defined in the requirements, or to detect possible problems which will have to be solved during later stages of the design. An interactive simulation can effectively lead to some questions that the designer will have to answer at a later stage.

Many tools (*ELUDO*, *LOLA*, *SMILE*, and *CAESAR*) allow the simulation of LOTOS specifications. During our research, we concentrated on the *ELUDO-XELUDO* and *LOLA* tools. The next three examples present the usefulness of the simulation in early stages of the design process.

Simulation of a Complete Specification Using XELUDO



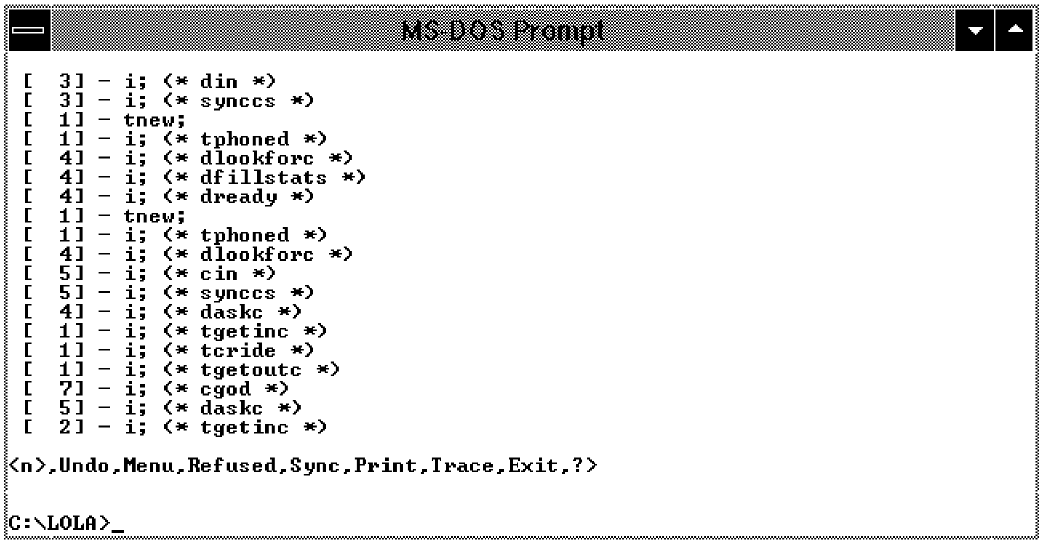
```
Traveler_Example[Tnew, Tdest]()
 Tnew;
  hidden Pready;
   hidden Cin;
    hidden Din;
     hidden TphoneD;
      hidden DlookforC;
       hidden DaskC;
        hidden Dfillstats;
         hidden Dout;
          hidden TgetinC;
           hidden TCride;
            hidden TgetoutC;
             hidden Cgarage;
              hidden Cout;
               hidden Tairport;
                hidden TgetonP;
                 hidden TPflight;
                  hidden TgetoffP;
                   hidden Phangar;
                    Tdest;
```

Figure 63: Simulation of Traveler_Example using XELUDO

We can simulate a simple scenario in a level 1 complete specification by using the tool XELUDO, an X-Windows based version of ELUDO. Figure 63 presents the history window of a *Traveler_Example* simulation. This trace shows that a traveler got to her destination, and then no more actions are possible (LOTOS deadlock). This level 1 specification leads to a deadlock because there is no recursion. Simulation helps designers validate the ordering of activities and their availability.

Simulation of a Complete Specification Using LOLA

LOLA also provides means to simulate specifications. In this example, we executed the level 3 specification *Traveler_Level3* to show that we can use any level of specification, even mixed-level specifications. The trace of figure 64 raises a few questions that cannot be raised using a level 1 specification.



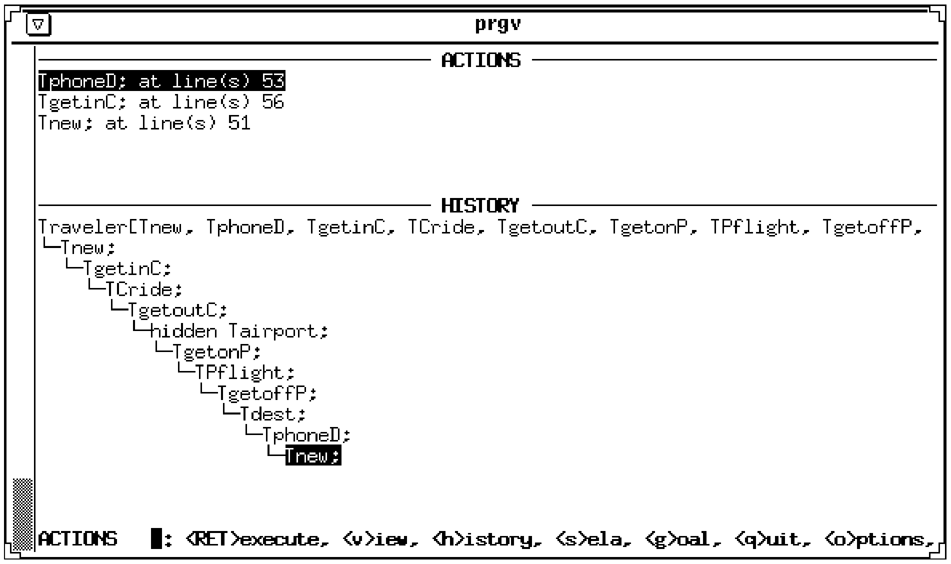
```
 [  3] - i; (* din *)
 [  3] - i; (* synccs *)
 [  1] - tnew;
 [  1] - i; (* tphoned *)
 [  4] - i; (* dlookforc *)
 [  4] - i; (* dfillstats *)
 [  4] - i; (* dready *)
 [  1] - tnew;
 [  1] - i; (* tphoned *)
 [  4] - i; (* dlookforc *)
 [  5] - i; (* cin *)
 [  5] - i; (* synccs *)
 [  4] - i; (* daskc *)
 [  1] - i; (* tgetinc *)
 [  1] - i; (* tcride *)
 [  1] - i; (* tgetoutc *)
 [  7] - i; (* cgod *)
 [  5] - i; (* daskc *)
 [  2] - i; (* tgetinc *)

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?>

C:\LOLA>_
```

Figure 64: Simulation of Traveler_Level3 using LOLA

From this simulation, the designer can see that the dispatcher, after receiving two requests from two travelers (`tphoned`), finally finds a taxi (`daskc`). The designer could wonder what was his initial intention here: how many requests can the dispatcher accumulate before he tells the travelers he cannot take any more requests? Is there any means for the dispatcher to tell the next travelers that they would have to call back later, when the system allows it? In the first loop of timethread *Dispatcher*, is it normal that the dispatcher fills his statistics (`dfillstats`) without having any news from the first taxi? Also, following the semantics we gave to the timethreads, a taxi can only take one traveler. Is that what we really intended? Should we specify a maximum number of travelers (say 3) that a taxi can take in? Does the same thing happen with travelers and planes? All these questions could be raised by executing a simple sequence from a timethread diagram. These issues would have to be solved in some way during the later stages of the design process (at the architecture level).

<u>Simulation of a Single Process Using ELUDO</u>

Most tools allow the simulation of single processes and compositions of processes. This functionality provides the opportunity to simulate single timethreads and topologies of interacting timethreads. In the next example (fig. 65), we use ELUDO to simulate a single process (*Traveler*, level 3).



```
prgv
------------------------- ACTIONS -------------------------
TphoneD; at line(s) 53
TgetinC; at line(s) 56
Tnew; at line(s) 51


------------------------- HISTORY -------------------------
Traveler[Tnew, TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPflight, TgetoffP,
└─Tnew;
  └─TgetinC;
    └─TCride;
      └─TgetoutC;
        └─hidden Tairport;
          └─TgetonP;
            └─TPflight;
              └─TgetoffP;
                └─Tdest;
                  └─TphoneD;
                    └─Tnew;

ACTIONS   : <RET>execute, <v>iew, <h>istory, <s>ela, <g>oal, <q>uit, <o>ptions,
```

Figure 65: Simulation of process Traveler using ELUDO

We observe (fig. 65) that the activity *TphoneD* occurred after *Tdest*. This means the dispatcher received the phone call after the traveler arrived to her destination! What is misleading here is the absence of two interacting timethreads: *Dispatcher* and *Cab*. They would have constrained (refer to figure 15) the event *TphoneD* to occur before *TgetinC*, and thus before *Tdest*.

This example shows that the simulation of single timethreads, although useful in some cases, is less interesting than the execution of complete timethread maps because the context (other interacting timethreads) is absent. This sometimes results in traces difficult to understand. However, once this limitation is understood, the traces can become meaningful.

5.4.2 Testing

Different testing strategies based on LOTOS can be used to validate a timethread map (see [Led 91] and [Mye 79] for further development). In our case, we suggest some sort of *design testing* where we execute a high-level specification (or play the design) with the intent of finding errors or problems. The different executions are called *test cases*. “A good test case is one that has a high probability of detecting an as-yet undiscovered error” [Mye 79].

Although we will use design testing later on to test conformance between original and transformed specifications, this must not be considered as *conformance testing*. The latter aims at testing if an implementation conforms to a specification. This topic is not covered in this thesis.

One way to apply a design testing strategy in LOTOS is the composition of *acceptance* and *rejection* test cases with our non-deterministic specifications.

An acceptance test represents a valid scenario that the timethread map and its corresponding LOTOS specification have to be able to execute (without deadlock). The map and its specification must not be able to execute rejection tests, which are invalid scenarios. These test cases can be derived from the requirements or from previous timethread maps and specifications.

There are many similarities between this technique and another one called *grey-box testing*, used to test design representations [Pro 92]. We already have a design represented as a precise sequence of inputs, decisions, and actions in a testable form, in this case a timethread map and its LOTOS specification. We also use a black-box testing technique (LOTOS testing based on acceptance and rejection). Some elements are however missing in our case in order to have a complete grey-box testing technique:

- We need the definition of a set of test attributes for timethread maps;
- We need ways to measure design coverage;
- We need heuristics for the generation of test cases.

These three missing elements are research issues. Once the test cases are available, we can use different LOTOS validation tools (such as LOLA or ELUDO).

Testing of a Level 1 Specification Using LOLA

For our *Traveler System*, we can use the map of figure 14 to derive one acceptance test case and one rejection test case (spec. 8). We assume here we will apply them to the level 1 specification *Traveler_Example*. The test `AcceptTest1` expresses that the system has to accept the a new traveler (`Tnew`) and lead her to the destination (`Tdest`). This is required from the use case of figure 14. `RejectTest1` tests that, at a level 1, a traveler gets to her destination before the system allows another new traveler.

```
(* Accept Test Case for level 1 *)
    process AcceptTest1[Tnew, Tdest, Success] : noexit :=
        Tnew;
        Tdest;
        Success; stop
    endproc (* AcceptTest1 *)

(*-------------------------------------------------------*)

(* Reject Test Case for level 1 *)
    process RejectTest1[Tnew, Tdest, Success] : noexit :=
        Tnew;
        (* There should not be a second Tnew possible *)
        Tnew;
        Tdest;
        Success; stop
    endproc (* RejectTest1 *)
```

Specification 8: Test cases for level 1 Traveler System

If we had developed intermediate designs and maps, we could have used them to derive a more complex test suite.

In LOTOS, test cases are represented as processes. They are then composed (synchronized) with the system. An acceptance test passes if the composition reaches the end of the test case (here, we use the gate *Success*). A rejection test passes if the composition does not reach the end of the test case.

LOLA implements this black box testing methodology. It follows the definition of *Testing Equivalence* of de Nicola and Hennessy, and it outputs one of the three following results:

- **`MUST PASS`** (*Must test*): Given a specification *L* and a test *T*, *T* is a must test of *L* if it terminates for every execution of the system when applied to *L*.
- **`MAY PASS`** (*May test*): Given a specification *L* and a test *T*, *T* is a may test of *L* if it terminates for at least one execution of the system when applied to *L*.
- **`REJECT`** (*Reject test*): Test which is neither may or must, i.e., no execution terminates successfully.

To compose these test cases with the specification, we use the LOLA command *TestExpand*. TestExpand makes a complete state exploration and calculates the type of response (must, may or reject). It needs three arguments: the depth of search in the labeled transition system (-1 means no limit), the success event, and the test process. The acceptance test gives the following result:

```
lola> TestExpand -1 Success AcceptTest1
 Composing behaviour and test :

    Analysed states       = 12
    Generated transitions = 15
    Duplicated states     = 0
    Deadlocks             = 0

    Process Test = accepttest1
    Test result  = MUST PASS.

                   successes = 4
                       stops = 0
                       exits = 0
               cuts by depth = 0
```

Our acceptance test case is a must test. Therefore, our timethread map is consistent (test equivalent) with the use case of section 3.3.1.

We can also compose our reject test case with the specification. LOLA outputs:

```
lola> TestExpand -1 Success RejectTest1

 Composing behaviour and test :

    Analysed states       = 8
    Generated transitions = 7
    Duplicated states     = 0
    Deadlocks             = 4

    Process Test = rejecttest1
    Test result  = REJECT.
```

```
       successes = 0
           stops = 4
           exits = 0
   cuts by depth = 0
```

All possible traces led to deadlocks, and the *Success* event could not have been reached. The rejection test was rejected as it was supposed to be. Therefore, our specification behaved properly.

Testing of a Level 3 Specification Using LOLA

Another test suite that assesses what the level 3 specification of our system must accept and reject is presented in specification 9.

```
(* Accept Test Cases for level 3 *)
    process AcceptTest1[Tnew, Tdest, Success] : noexit :=
        Tnew;
        Tdest;
        Success; stop
    endproc (* AcceptTest1 *)

    process AcceptTest2[Tnew, Tdest, Success] : noexit :=
        Tnew;
        (* This time, because of the recursion, a second *)
        (* traveler is allowed *)
        Tnew;
        Tdest;
        Success; stop
    endproc (* AcceptTest1 *)

(*-------------------------------------------------------*)

(* Reject Test Case for level 3 *)
    process RejectTest1[Tnew, Tdest, Success] : noexit :=
        Tnew;
        Tdest;
        (* A traveler cannot get to her destination *)
        (* before leaving! *)
        Tdest;
        Success; stop
    endproc (* RejectTest1 *)
```

Specification 9: Test cases for level 3 Traveler System

TestExpand performs a state exploration of the composition of the system under test with the test case. If the number of states is infinite (this happens with level 3 specifications), then we cannot use this command. For this reason, LOLA provides another command (*OneExpand*) that executes random traces of the composition. OneExpand has four important arguments: the depth or search, the success event, the test process, and the seed (used for random number generation).

Because OneExpand does not explore all possible states, we have to apply our test cases many times, using different seeds, to insure the validity of the result. In our example, we limited to five the number of times each of our three tests is executed. Different seeds (4, 7, 11, 13, 17) were used.

The results of the three following sets of tests are cumulated in figure 66. Note that we limited to 100 events the depth of the search of the rejection test case. For a result from a rejection test to be meaningful, we have to be sure the depth is high enough.

```
lola> OneExpand -1 Success AcceptTest1 Seed
lola> OneExpand -1 Success AcceptTest2 Seed
lola> OneExpand 100 Success RejectTest1 Seed
```

Test case	Seed	Result	Transitions generated
AcceptTest1	4	SUCCESSFUL EXECUTION	48
AcceptTest1	7	SUCCESSFUL EXECUTION	37
AcceptTest1	11	SUCCESSFUL EXECUTION	54
AcceptTest1	13	SUCCESSFUL EXECUTION	65
AcceptTest1	17	SUCCESSFUL EXECUTION	57
AcceptTest2	4	SUCCESSFUL EXECUTION	44
AcceptTest2	7	SUCCESSFUL EXECUTION	82
AcceptTest2	11	SUCCESSFUL EXECUTION	55
AcceptTest2	13	SUCCESSFUL EXECUTION	82
AcceptTest1	17	SUCCESSFUL EXECUTION	82
RejectTest1	4	REJECTED EXECUTION	100
RejectTest1	7	REJECTED EXECUTION	100
RejectTest1	11	REJECTED EXECUTION	100
RejectTest1	13	REJECTED EXECUTION	100
RejectTest1	17	REJECTED EXECUTION	100

Figure 66: Results of testing (Traveler_Level3) using LOLA

From these results, we can conclude with a certain degree of confidence that the two acceptance tests are *must* tests, and the rejection test is a *reject* test. Therefore, the designer has more confidence in his system to be valid w.r.t. the use case and the requirements.

Many more interesting and useful test cases could be generated for a specification where all activities are observable. When we make an activity internal or hidden in LOTOS, we cannot test it afterwards by using traditional LOTOS black-box testing because the test case cannot synchronize on the corresponding gate. By making everything visible, we obtain a grey box in which all cause-effect relations can be tested. This feature is desirable for design testing and grey-box testing [Pro 92], and our method allows it.

5.4.3 Verification

This third validation technique aims at demonstrating consistency between two designs. It also helps proving desirable properties of a system such as the absence of deadlock. Verification is usually very costly (and sometimes impossible) because it implies exhaustive search in a huge number of system states. ELUDO and LOLA are not powerful verification tools, but we can still use them to verify a few properties.

Verification of a Level 1 Specification Using LOLA

The best verification feature of LOLA is the command *TestExpand* presented in the previous section. TestExpand verifies the testing equivalence between two processes (or two designs). Other equivalence relations (bisimulation, trace equivalence...) exist and can be verified with other tools (such as *Squiggle*). Although what we really need is a definition of *path equivalence*, testing equivalence fills most of needs of timethread maps verification.

Another interesting feature of LOLA is the expansion function *Expand*. This command calculates the EFSM (Extended Finite State Machine) of a behaviour or a specification. This EFSM can be used as an underlying model for model checking, with other tools. It can also help in finding deadlocks in a specification. The following example shows that the EFSM of the level 1 specification *Traveler_Example* has 14 deadlocks. The latter are due here to the fact that we deal with a specification without recursion. In other words, such deadlocks were explicitely included in the specification (**`stop`**) by the designer. If they occur in a recursive specification (as in a non-stopping RTD system), then this indicates a design problem.

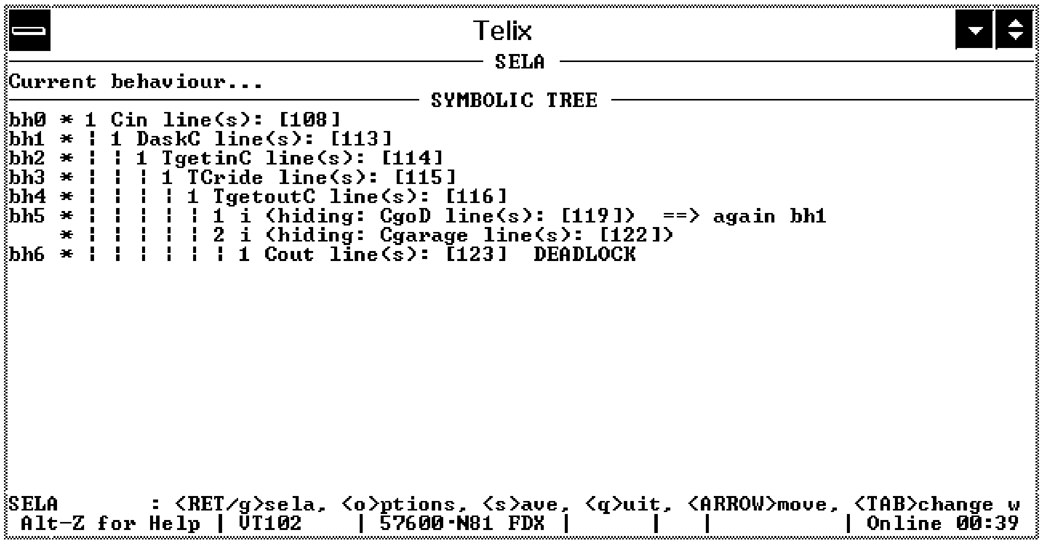
```
lola> Expand -1

    Analysed states       = 221
    Generated transitions = 539
    Duplicated states     = 319
    Deadlocks             = 14
```

The automaton obtained can be normally checked to verify that the behaviour is as desired in all cases. In practice, the goal can be daunting, as it can be seen from this example, where a very large number of states and transitions were obtained from a small specification. A recursive specification may also lead to an infinite number of states. In that case, the EFSM is truncated and the specification is partially verified only.

Verification of Process Cab Using ELUDO

ELUDO includes a tool called *SELA*, which performs the symbolic expansion of a process or a specification. The output is a tree-like structure that shows all possible traces of events. This tree can help the designer finding undesirable sequences of events. This process is very similar to LOLA's expansion.

An example of symbolic tree output from SELA is shown in figure 67. The process *Cab* (level 1) has been expanded and all possible sequences are presented in the tree. Undesirable deadlocks can be found in this way.



```
                                 SELA
Current behaviour...
                              SYMBOLIC TREE
bh0 * 1 Cin line(s): [108]
bh1 * ¦ 1 DaskC line(s): [113]
bh2 * ¦ ¦ 1 TgetinC line(s): [114]
bh3 * ¦ ¦ ¦ 1 TCride line(s): [115]
bh4 * ¦ ¦ ¦ ¦ 1 TgetoutC line(s): [116]
bh5 * ¦ ¦ ¦ ¦ ¦ 1 i (hiding: CgoD line(s): [119])  ==> again bh1
    * ¦ ¦ ¦ ¦ ¦ 2 i (hiding: Cgarage line(s): [122])
bh6 * ¦ ¦ ¦ ¦ ¦ ¦ 1 Cout line(s): [123]  DEADLOCK










SELA       : <RET/g>sela, <o>ptions, <s>ave, <q>uit, <ARROW>move, <TAB>change w
 Alt-Z for Help | VT102    | 57600-N81 FDX |      |   |         | Online 00:39
```

Figure 67: Verification of process Cab using SELA

Other Verification Methods

Large symbolic expansion graphs such as those obtained from LOLA or SELA can be inspected automatically by using a procedure called *model-checking*. The graph is transformed into a Kripke structure on which we perform model checking. Properties to be checked can be expressed using a temporal logic formula. *LMC* [Ghr 92] is a LOTOS model checker that can verify properties expressed in the temporal logic language CTL. The tool CAESAR also provides a model checker (called *ALDEBARAN*).

Other verification tools using goal-oriented execution, trace theory, or equivalence relations could be integrated into a timethreads-LOTOS verification environment.

CHAPTER 6

Case Study: *Telepresence* A Multimedia System Design Example

6.1 Introduction

“*Telepresence is a set of computer, audio-video and telecommunications technologies, which are carefully integrated to enable people to work together using technology as an intermediary. Properly deployed, telepresence conveys in users the feeling of being present in each other's offices from remote distances. More than simple video-conferencing, telepresence attempts to duplicate the subtle social protocols, degrees of confidentiality, intimacy and trust in everyday relationships and interactions when remote persons are brought together.*” [Man 93].

Telepresence represents a very complete multimedia system. The complexity of such real-time and distributed system makes its design a true challenge. In this chapter, we will use our techniques based on the Timethread notation and the formal language LOTOS to help specifying, refining, and validating a first design.

A designed system must be reliable and reusable as much as possible. This case study mostly deals with reliability concerns. Reusability, expressed in terms of role architectures, frameworks, class hierarchies, and objects [Buh 93] will not be covered in order to simplify the problem.

The *Telepresence* system presented in section 6.2 is designed from a user-centered viewpoint. Many assumptions are made to simplify its complexity. This system is neither a complete telepresence system nor an existing one. We only use ideas from the telepresence project to get interesting and challenging scenarios allowing us to use our approach. In section 6.3, the LOTOS interpretation method for timethreads is applied to the example in order to get the first path specification from the first timethread map. Section 6.4 presents the factoring transformation leading to a second timethread map and its corresponding LOTOS specification. We also overview the validation and consistency of the two specifications.

6.2 The Telepresence System

6.2.1 General Telepresence System

A complete *Telepresence* system might offer many different types of services to simulate virtual presence and other facilities:

- Electronic mail.
- VoiceMail (using a computer mail program or using the phone).
- VideoMail.
- Visual contact.
- Teleconference.
- Data exchange.
- Desk Automated Network (for instance, a group of people working on the same document), where everything is integrated/distributed.
- Receptionist (Virtual Automated Attendant), etc.

Many hardware and software components are usually required to provide such services:

- Users' workstations (PC, Macintosh, Sun...) with a lot of RAM,
- Color monitors (possibly more than one per user),
- Microphones (e.g., a small PZM microphone),
- Speakers (with volume control),
- Color camera unit (usually small, possibly more than one per user),
- Phones,
- Different high-speed networks,
- Servers' workstations (Sun...),
- Audio/Video switching devices,

- VCRs,
- Codecs (Compressors/Decompressors, Picturetel units),
- Integrated Interactive Intermedia Facility (IIIF Server, software) [Mil 92],
- Telepresence Communication Server (TCS, software) [Man 93],
- Voice Server (software) [Jac 93], etc.

We can have many different views of a telepresence system, and we will concentrate on only one of them. A user-centered design is a viewpoint where the functions provided by the system need to be those functions that will fulfill the communication needs of the user. This design can be achieved by specifying, for example, the user interface. "*The design of the user interface serves as the driving thrust for the development focus of Telepresence teams*" [Man 93]. We use this approach in order to get a first LOTOS "prototype" allowing us to play the design (by executing the specification) and validate it against the requirements.

The use of many simplification assumptions will help us in concentrating with the real problem, without committing to detailed solutions too soon. We do not aim at designing a complete and real *Telepresence* system in this chapter. A simple system is complex enough to illustrate the method and design issues.

6.2.2 Informal Description and General Assumptions

This case study emphasizes on the *visual contact service*, which can be considered as a special application of the teleconferencing capabilities of a complete *Telepresence* system. To simplify further, we will constrain our *Telepresence* system, from a user's perspective, to the basic components (names and icons) presented in figure 68:

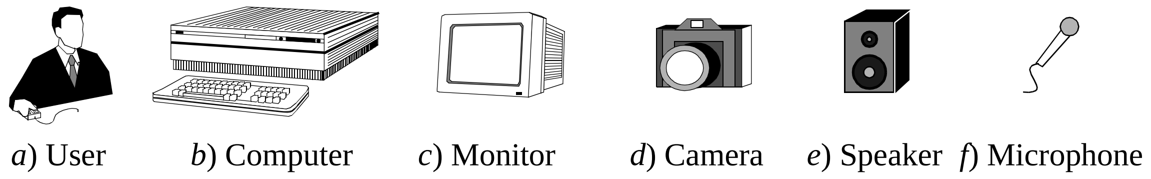


Figure 68: Basic components

In order to describe the functionalities of the system, two concepts have to be introduced:

Initiator-Responder concept: The user who initiates a connection is called the *initiator*, and the one who is being contacted is the *responder*. The system must provide the correct functionalities to satisfy a user who can be in any (or both) of these roles.

Door concept: A *door* is a means to protect a user's privacy. An initiator can look into a responder's office only when the responder's door is open. A responder can open or close his/her door at anytime. This is a simplified definition of the door concept introduced in the real telepresence project [Man 93], where a complex protocol is needed to provide flexibility in the privacy management. In our *Telepresence* system however, a user will not be allowed to lock the door, to glance at one's door, to select users for whom the door is always open or closed, etc.

The *visual contact service* functionalities can be informally described, from a user's viewpoint, in the following way:

Initiator: During the connection phase, an initiator sends a *Contact* request to a responder, then he/she waits for a *Report* telling whether the access was successful or denied, or if a timeout occurred (when the request is lost somewhere in the system). If the contact is established, the transmission phase starts and the initiator receives images on his/her monitor and his/her speakers play the responder's voice. The initiator can *Terminate* the transmission (and the connection) at any time after the transmission has started, and then he/she is allowed (*Next*) to contact another responder. An initiator can contact only one responder at a time.

Responder: At any time, a responder can *Open* or *Close* his/her door to allow or deny access to initiators' connection requests. A signal appears on the responder's monitor as long as an initiator is looking into the responder's office. During this transmission phase, the camera and the microphone record images and voice to be transmitted.

Any user can be initiator and/or responder. Also, a responder can contact a third user, not necessarily the one looking in his/her office. There is no constraint on the number of initiators who can look into a specific responder's office.

In section 6.3, we will describe this *Telepresence* system as two different entities: an initiator system under design (SUD) and a responder SUD. Later, the two systems will be merged into one complete system in order to allow peer-to-peer relationship, i.e., a complete system *A* can be initiator and a complete system *B* responder, and vice-versa. This approach is similar to the *Message Transfer Unit* example developed in [Buh 93].

Although we are able to describe a few properties and functionalities of the system, we do not intend to formulate a detailed specification, neither with timethreads nor with LOTOS. We will get a "projection" of the end-to-end paths onto LOTOS behaviour and processes, and we will have to get as much information as possible from these specifications and diagrams.

6.3 The First Timethread Map

The approach consists of a timethread-centered design, leading to a timethread map interpreted as a LOTOS specification for simulation, refinement, and validation. We will try to think of this method as a CAD tool, i.e., with a timethread graphical editor, automated translation of a timethread map into LOTOS, automated support of transformations, and other functionalities for the validation[1]. Of course, no such integrated tool exists yet, but the design process needs this type of tool, and therefore we think we should get used to this way of thinking.

6.3.1 General View of the Approach

We use here an approach based mostly on the timethread-centered design and on the LOTOS interpretation method for timethreads. We intend to:

- Define the basic components (as few as possible, only pertinent ones).
- Build use cases, using timethreads, for the description of individual scenarios.
- Combine and complete those timethreads to form end-to-end paths in our first timethread map.
- Perform the LOTOS interpretation method for timethreads in order to get the first specification.
- Use the specification as a means to "play the pictures", for partial validation against the requirements.

The first step was already covered in section 6.2.2 where the basic components were identified (see figure 68).

1. A specification is generally validated against previous specifications or the requirements.

6.3.2 Basic Scenarios

We present here a few steps involved in the production of a first timethread map. Specific scenarios are investigated and some design decisions must be taken. This is one of the many possible ways leading to timethreads discovery, and the pertinence of the resulting map is outside the scope of this thesis. However, we try to get a map as meaningful and realistic as possible for the sake of the case study.

The following scenarios are grouped into three categories, more or less expressed in the requirements: connection phase, transmission phase, and door management.

Connection Phase

The timethread in figure 69a shows a very simple use case [Jac 93] of the initiator’s connection phase. This high-level scenario shows a sequence of interactions between the user in an initiator role and the *Telepresence* system. When a *Contact* request is sent, the resulting event is a *Report*. Such use case, although very simple, is useful in early stages of the design process (especially when we start from a “blank sheet”) to understand simple cause-to-effect relationships without committing to any internal detail. Simple test cases can also be derived from such descriptions. However, no more use cases will be shown in this chapter for space reasons.

“An initiator can contact only one responder at a time” means that a user cannot initiate more than one telepresence session at the same time. We therefore have to constrain the number of instances of the *Connection* timethread during the connection phase. This is expressed in figure 69b, where the *Next* event indicates that a new connection can be established. Note that we added an AND-Fork to indicate that the initiator can look at the report whenever he/she wants.

The requirements indicate three types of reports: *Success*, *Denied*, and *TimeOut*. We can use tags to tell the initiator which internal path has been taken (fig. 69c). The addition of these alternate paths is an instance of extension, as introduced in section 5.3.3.

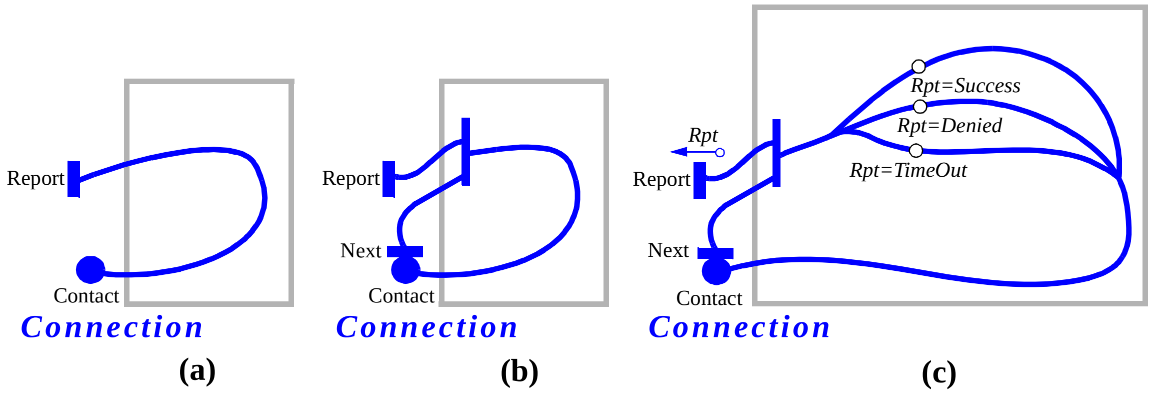


Figure 69: Connection phase

We know that, during the connection phase, the responder's door has to be checked. The timethread *Knocking* (fig. 70a) shows one possible sequence. Also, the resulting event (here *DoorChecked*) has to report the accessibility of the responder (tag values *Access* or *Denied*). We use a tag mechanism (fig. 70b) to manage the flow of information from timethread *Knocking* to the timethread to which it will somehow be connected (in occurrence, *Connection*). Note that this timethread and its activities are internal to the system, i.e., hidden from the users.

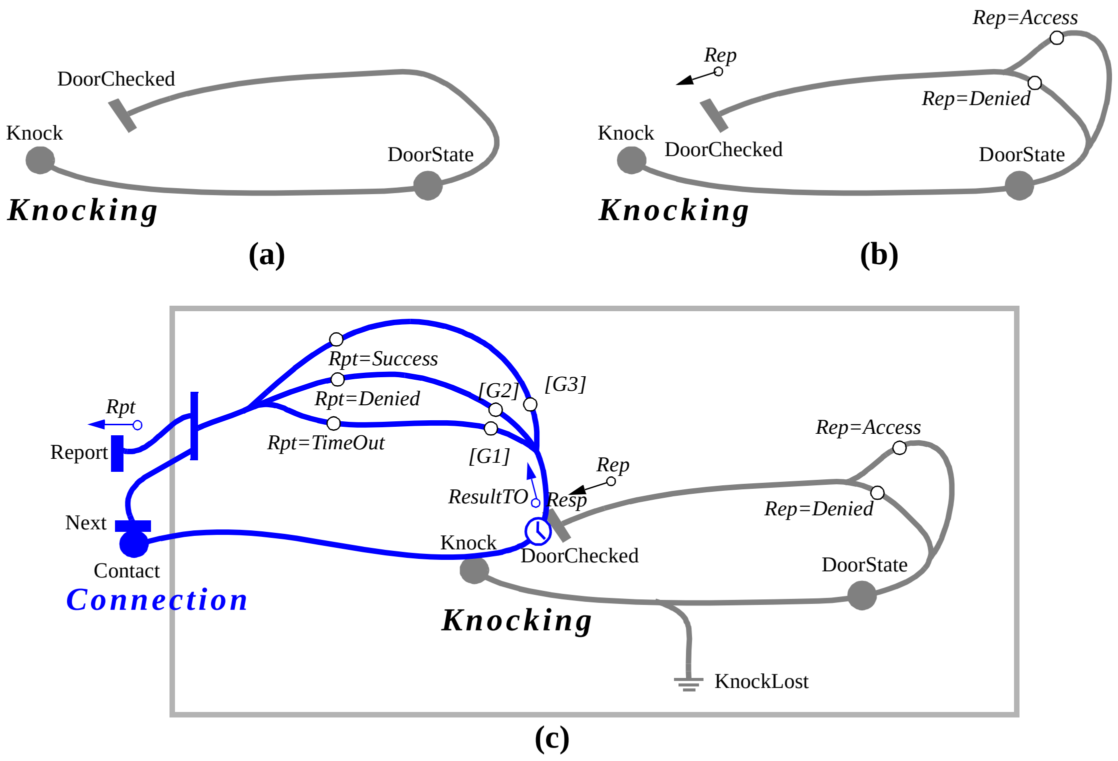


Figure 70: Connection phase and Knocking sequence

Figure 69c presents the complete connection phase, which is a composition of the two timethreads already developed. A *Contact* request will cause (in passing) a *Knock* event in the *Knocking* timethread. The result, returned in *DoorChecked*, will cause a time waiting place to stop waiting, without a timeout, and the tag *Resp* will be available to the guards *G1* to *G3* (see §6.3.3 for a description of the guards). If the *Knocking* request is lost, then a timeout will occur and the internal *TimeOut* (time waiting place) will be performed, causing *Report* to tell the user about the timeout. We added the *KnockLost* internal event to test robustness in our design.

Transmission Phase

The transmission phase is first represented by the simple *Transmission* timethread of figure 71. When an internal event *Transmit* occurs, the *Signal* is displayed on the responder's monitor and this causes the microphone to record the voice (*RecordVoice*) and, concurrently, the camera to record the image (*RecordImage*). The initiator's monitor then displays the image (*PlayImage*) while the speaker outputs the voice (*PlayVoice*). Then, either the transmission continues or the *Disconnect* activity is performed, causing the end of reception (*EndRec*).

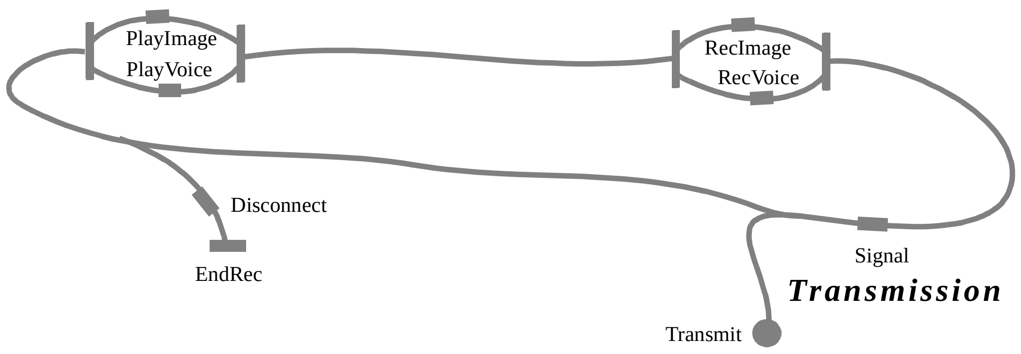


Figure 71: Transmission phase

In the requirements, the initiator can *Terminate* the transmission, therefore the disconnection does not have to remain as non-deterministic as in the previous figure. In figure 72, we added a *Disconnection* timethread managing this event. It is connected to a waiting place *RecState* on the timethread *Transmission*. The waiting places described in the Timethread notation do not have any options corresponding to what we want to represent here. We do not want the *Transmission* timethread to always wait for an event or for a timeout.

At this point, we feel the need for a new type of waiting place that waits a synchronization event and then outputs a corresponding tag (say *Sig*, with the value *Yes*), or continues if no synchronization occurred and then outputs the tag with another value (*No*).

To this end, we introduce the new symbol Ⓢ to denote such a new type of waiting place, which we call **`Signal`**.

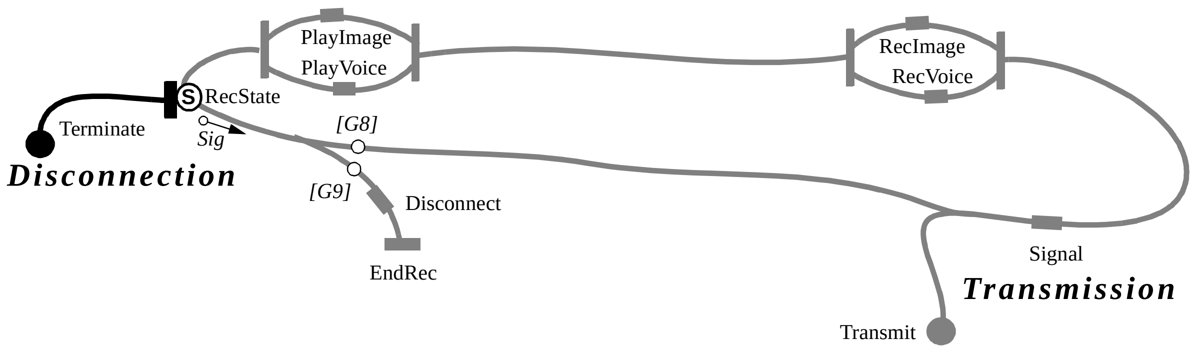


Figure 72: Transmission phase and disconnection management

The addition of `Signal` as a waiting place option (`<WPOptions>`) shows the extensibility power of the STDL grammar:

```
<WPOptions> = [<Delayed> | <Timed> | <Signal>]
```

A LOTOS interpretation of a generic *signal waiting place* could be:

```
(* Waiting place signal management *)
hide SyncSig in
(
   EventName; SyncSig ! Yes; stop  (* Synchronization *)
   []
   SyncSig ! No; stop              (* No synchronization *)
)
|[SyncSig]|
SyncSig ? Sig: Tag;
```

The tag *Sig* can then be used to constrain forthcoming choices, as with the guards *G8* and *G9*.

Looking at the previous figure, we see a strong coupling between the information recorded and the information played, i.e., we cannot record more than one image before it is played. We could loosen this coupling by splitting the *Transmission* timethread into two parts, one managing the transmission and the other (*Reception*) managing the reception. This is shown in figure 73, where the two timethreads communicate via two internal events: *Receive* and *SendState*. Note that the implementation of the waiting place *Receive* might require the existence of a buffer (or a queue) between *Reception* and *Transmission*, unless decided otherwise at a later design stage, perhaps at the architecture level.

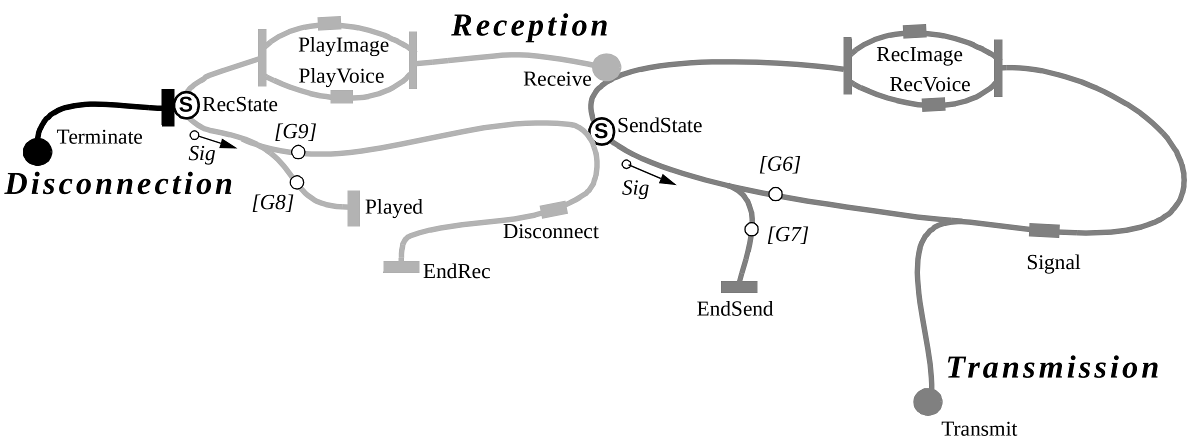


Figure 73: Transmission phase - Decoupling

Again, we used a signal waiting place to decide, with the help of guards *G6* and *G7*, whether or not other information has to be transmitted. Two new internal results were added to complete the timethreads: *Played* and *EndSend*.

<u>Door Management</u>

The requirements also indicate that a responder can *Open* or *Close* his/her door at any time. This is expressed by the two timethreads of figure 74.

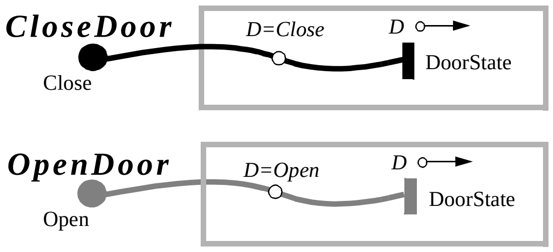


Figure 74: Door management (Open and Close)

These timethreads both result in updating the door state either with the value *Open* or with the value *Close*.

6.3.3 First End-to-end Paths Timethread Map

In the previous section, we discovered the basic functionalities of the *Telepresence* system from a user interface viewpoint. We constructed our timethreads following the connection and transmission phases, and the door management. We can compose our timethreads to form the first timethread map representing end-to-end paths of our *Telepresence* system (figure 75).

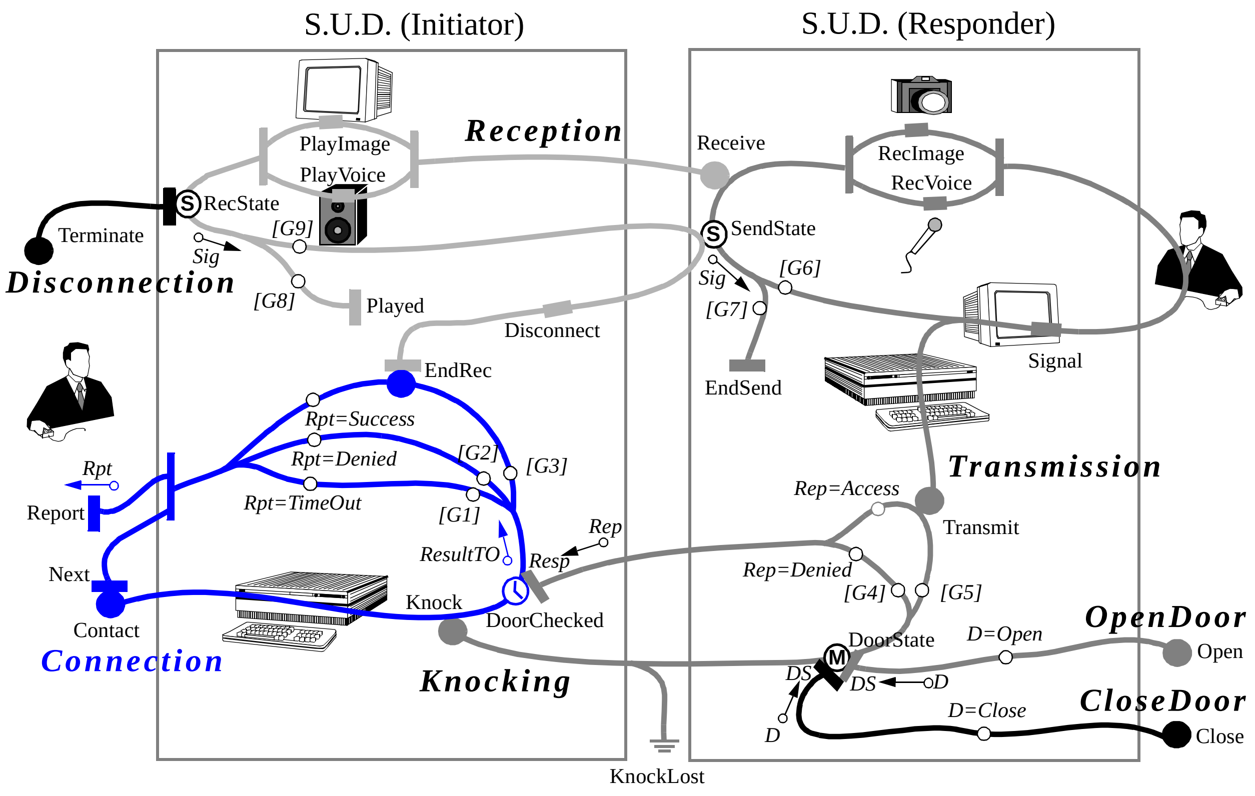


Figure 75: First Telepresence timethread map (end-to-end paths)

This timethread map captures the big picture of the system. It is not concerned with details like protocols, data, transmission of data, control of hardware, etc. Only interesting paths along relevant components are shown here. Note that, although they are not explicitly drawn on the diagram, we assume that the initiator SUD and the responder SUD both have cameras, speakers, and microphones.

Many steps have been involved in the generation of this timethread map. The most important ones are:

- The separation of the Initiator SUD from the Responder SUD.
- The connection of *Reception* and *Connection* on the waiting place *EndRec*.

- The connection of *Knocking* and *Transmission* on the starting event *Transmit*.
- The connection of *Knocking*, *OpenDoor* and *CloseDoor* on the Memory waiting place *DoorState* (see below for the definition of such waiting place).
- The creation and update of guards *G1* to *G9* based on the different tags. Their definitions are:

```
[G1] is [ResultTO eq TOut]
[G2] is [(ResultTO eq OK) and (Resp eq Denied)]
[G3] is [(ResultTO eq OK) and (Resp eq Access)]
[G4] is [DS eq Close]
[G5] is [DS eq Open]
[G6] is [Sig eq No]
[G7] is [Sig eq Yes]
[G8] is [Sig eq No]
[G9] is [Sig eq Yes]
```

We extended again our notation with the creation of a new type of waiting place called **`Memory`**. This is reflected in the STDL grammar with the addition of a new waiting place option (`<WPOptions>`) also named `Memory`. The purpose of a *memory waiting place* is to act as a variable, or a buffer, local to a waiting place. It has a default value, and it always provides its most recent value to a tag that can be used later to determine choices along the timethread path. It can also synchronize anytime with other timethreads, so no one really has to wait. The given LOTOS interpretation for a memory waiting place named *WPName* is:

```
process TimethreadName [TTGates..., WPName] : noexit :=
   (* hidden memory cell for internal use *)
   hide WPNameMem in
   WPNameMemory [WPName, WPNameMem] (InitValue)
   |[WPNameMem]|
   TimethreadName2[TTGates..., WPNameMem]
   where

   process TimethreadName2[TTGates..., WPNameMem] : noexit :=
      (* Timethread interpretation using WPNameMem instead of WPName *)
      ...
   endproc (* TimethreadName2 *)

   process WPNameMemory [WPName, WPNameMem] (Mem: Tag): noexit :=
      (* Process that can synchronize with external timethreads *)
      (* or with TimethreadName2 *)
      (* Get a new value *)
      WPName ? NewMem: Tag; WPNameMemory [WPName, WPNameMem] (NewMem)
      []
      (* Provide the current value *)
      WPNameMem ! Mem; WPNameMemory [WPName, WPNameMem] (Mem)
   endproc (* WPNameMemory *)
endproc (* TimethreadName *)
```

Again, this is an another example of the extensibility of the notation and the STDL grammar. Due to the modularity of the interpreted timethreads (1 timethread = 1 LOTOS process), such extensions do not result in complex LOTOS mappings.

6.3.4 Application of the LOTOS Interpretation Method

At this point, we consider our timethread map to be interesting enough to be mapped onto LOTOS. We therefore use the LOTOS interpretation method for timethreads.

Map Decomposition

The first step consists in deriving the LARG and the STDL descriptions from the timethread map. In figure 85, we present the STDL descriptions of the seven timethreads found in figure 75. The interactions between these timethreads are expressed in the LARG of figure 77. Again, we notice the one-to-one relationship between the timethreads and the processes.

Application of the LAEG Method

As usual, the LARG generated cannot be directly mapped onto LOTOS. Therefore, we apply the Grouping algorithm in order to get a binary grouped LARG (fig. 78). The solution we get is one among the hundreds possible equivalent solutions, but any of them can be use for validation. The structure generated can be found in the specification of Appendix C, lines 45 to 81.

```
Timethread Connection is
   Constrained
   Trigger (Contact)
   Async(Knock)
   Wait Time (Doorchecked ? Resp)
   Choice
         Guard (ResultTO eq TOut)  (* [G1] *)
         Tag (Rpt = TimeOut)
      Or
         Guard ( (ResultTO eq OK) and (Resp eq Denied) )  (* [G2] *)
         Tag (Rpt = Denied)
      Or
         Guard ( (ResultTO eq OK) and (Resp eq Access) )  (* [G3] *)
         Wait (EndRec)
         Tag (Rpt = Success)
   EndChoice
   AndFork
      Path Result (Report ! Rpt) EndPath
   EndAndFork
   Result(Next)
EndTT
```

```
Timethread Transmission is
   Internal Signal, RecImage, RecVoice
   Trigger (Transmit)
   Loop
      Comp
         Guard (Sig eq Yes)  (* [G7] *)
         Action (Signal)
         Par
               Action (RecImage)
            And
               Action (RecVoice)
         EndPar
         Async (Receive)
         Wait Signal (SendState)
      Opt
         Guard (Sig eq No)  (* [G6] *)
   EndLoop
   Result (EndSend)
EndTT
```

```
Timethread  Reception is
   Internal PlayImage,PlayVoice, Disconnect
   Trigger (Receive)
   Par
         Action (PlayImage)
      And
         Action (PlayVoice)
   EndPar
   Wait Signal (RecState)
   OrFork
         Guard (Sig eq Yes) (* [G9] *)
         Continue
      Or
         Guard (Sig eq No)  (* [G8] *)
         Path Result (Played) EndPath
   EndOrFork
   Async (SendState)
   Action (Disconnect)
   Result (EndRec)
EndTT
```

```
Timethread  Disconnection is
   Trigger (Terminate)
   Result (RecState)
EndTT
```

```
Timethread (Closing) is
   Trigger (Close)
   Tag (D = Close)
   Result (DoorState ! D)
EndTT
```

```
Timethread (Opening) is
   Trigger (Open)
   Tag (D = Open)
   Result (DoorState ! D)
EndTT
```

```
Timethread (Knocking) is
   Trigger (Knock)
   Loss (KnockLost)
   Wait Memory (DoorState ? DS)
   Choice
         Guard (DS eq Close)  (* [G4] *)
         Tag (Rep = Denied)
      Or
         Guard (DS eq Open)  (* [G5] *)
         Async (Transmit)
         Tag (Rep = Access)
   EndChoice
   Result (DoorChecked ! Rep)
EndTT
```

Figure 76: STDLdescriptions of the first Telepresence map

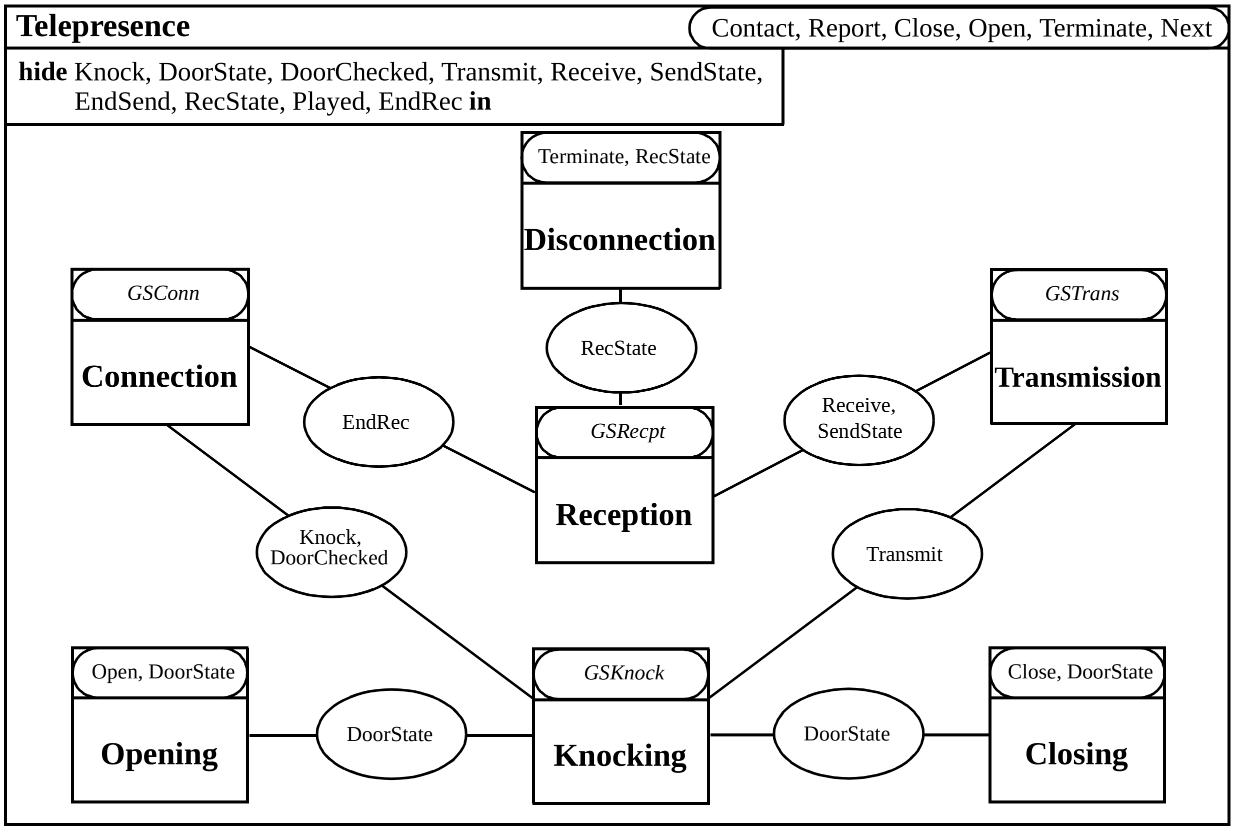


Figure 77: LARG of the first Telepresence map

Some interfaces (italicized label sets) were too lengthy to put in the diagram, but they are enumerated here:

- *GSConn* is `Contact, Knock, DoorChecked, EndRec, Report, Next`

- *GSKnock* is `Knock, DoorState, Transmit, DoorChecked`
- *GSRecpt* is `Receive, RecState, Played, SendState, EndRec`
- *GSTrans* is `Transmit, Receive, SendState, EndSend`

We find them again in the next LARG:

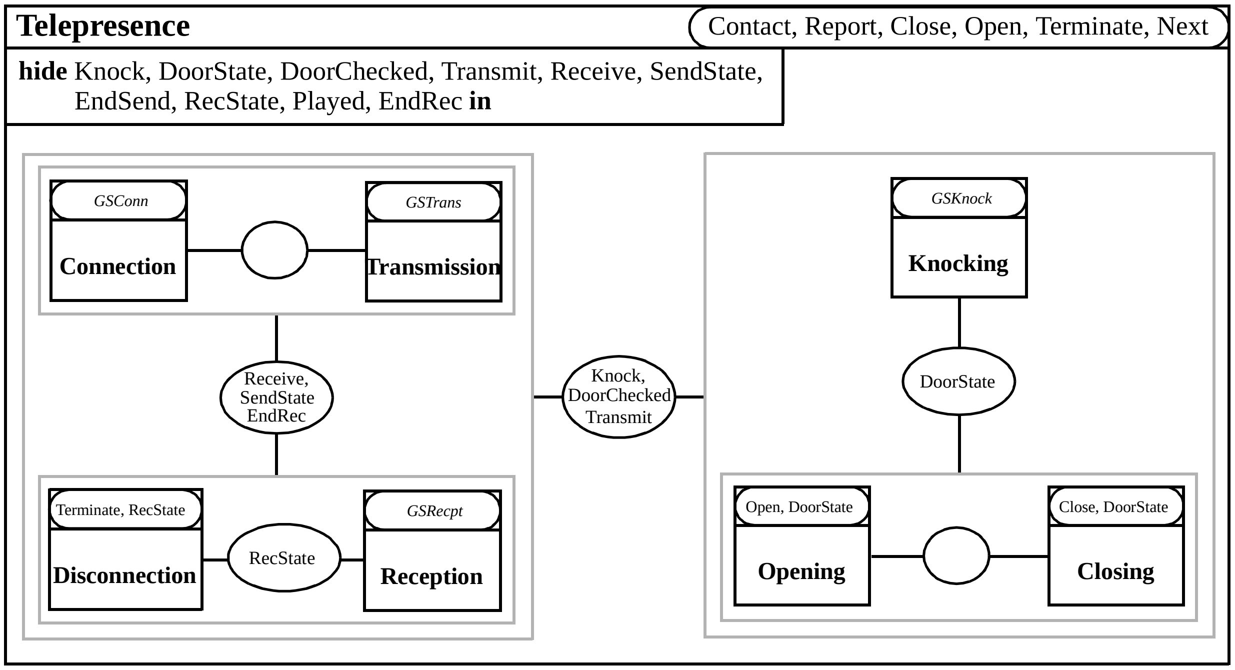

Telepresence
Contact, Report, Close, Open, Terminate, Next
hide Knock, DoorState, DoorChecked, Transmit, Receive, SendState, EndSend, RecState, Played, EndRec in
GSConn
Connection
GSTrans
Transmission
Receive, SendState EndRec
Terminate, RecState
Disconnection
RecState
GSRecpt
Reception
Knock, DoorChecked Transmit
GSKnock
Knocking
DoorState
Open, DoorState
Opening
Close, DoorState
Closing

Figure 78: Binary grouped LARG of the first Telepresence map

<u>Generation of Behaviour Expressions</u>

LOTOS processes are then generated following the STDL descriptions. Notice that we also interpreted the signal and memory waiting places found in the timethread map. In Appendix C, we found the level 1 (without recursion) interpretation of the *Telepresence* system.

As mentioned in section 4.5.5, mechanisms to ensure the availability of tags all along timethread paths are necessary. They are often needed for the synchronization points of the interpretation of STDL *Choice* and *Par* constructs. Instances of such internal tag flow are found in lines 114 to 126, and in lines 262 to 272 of the *Telepresence* specification. Also, since LOTOS requires all gates to have the same number and type of experiment offers (same number of tags in our case), dummy tags are needed on several occasions. For instance, in line 102, the second tag on gate `SyncTOCon` is a dummy tag; it is offered because other gates in the synchronization need two experiment offers. This dummy tag could have any value since it will not be used in guards.

To generate a level 1 specification with recursion, a few changes have to be introduced (fig. 79). The following twelve lines have to be slightly modified (modifications are *italicized*):

Stop becomes an off-end recursion of the calling process. Not all ***stop*** become recursive, only those following a resulting event:

```
134  Next; Connection[Contact, Knock, DoorChecked, EndRec, Report, Next]
177  [Sig eq Yes] (* [G7] *) -> (EndSend; Transmission[Transmit, Receive, SendState, EndSend])
189  RecState; Disconnection[Terminate, RecState]
228  EndRec; Reception[Receive, RecState, Played, SendState, EndRec]
231  [Sig eq No] (* [G6] *) -> (Played; Reception[Receive, RecState, Played, SendState, EndRec])
258  KnockLost; Knocking2[Knock, DoorStateMem, Transmit, DoorChecked]
274  Knocking2[Knock, DoorStateMem, Transmit, DoorChecked]
293  DoorState ! D; Opening[Open, DoorState]
304  DoorState ! D; Closing[Close, DoorState]
```

For process *TLoop* to be able to call *Transmission*, the gate *Transmit* has to be added:

```
145  Transmit; TLoop [Transmit, Receive, SendState, EndSend, Signal, RecVoice, RecImage]
148  process TLoop [Transmit, Receive, SendState, EndSend, Signal, RecVoice, RecImage]:noexit:=
175  (TLoop [Transmit, Receive, SendState, EndSend, Signal, RecVoice, RecImage])
```

Figure 79: Modifications to get a level 3 Telepresence specification

Composition of the Complete Specification

The simple composition procedure consists in adding the process definitions to the structure generated from the LARG. The final result is the level 1 specification *Telepresence* described in Appendix C.

6.3.5 Validation of the First Specification

We can now play the design and validate some functionalities against the requirements and use cases. For this purpose, we use validation techniques (§5.4) on a level 1 specification with recursion. We consider this level adequate for validation because:

- It is more realistic than a level 1 without recursion,
- A level 3, which is more complex, does not bring much enhancement since the *Connection* timethread is constrained to one instance at a time, and thus this constrains *Knocking* and *Transmission* also to one instance at a time.

We give examples of simulation and testing scenarios in Appendix D. We discuss the results here:

Simulation

We executed one sequence with ELUDO to look for possible problems with our first timethread map. Several problems were discovered, and we can relate them to the emphasized events (boxes 1 to 6 in Appendix D.I).

- The event *Terminate* was accepted before any transmission started (box 1), although the requirements specified that the disconnection should be available only after the transmission starts. The interface of our *Telepresence* system should not allow the use of *Terminate* at this point. This error leads to other problems also shown in the trace. For instance, box 3 indicates that the same transmission could be terminated twice by the initiator. This will affect a second transmission later on, as shown by box 6 where the second transmission is terminated while the first one was aimed. This is due to the cumulation of *Terminate* events, which is not appropriate at this point. The designer now knows that a mechanism is needed to ensure *Terminate* is not allowed, or not considered, before the beginning of the transmission, and that only one *Terminate* must be considered for a given connection.

- As indicated with box 2, images and voice data are not necessarily received in the right order. The timethread map and the *Receive* waiting place do not specify any ordering. It is the responsibility of the designer to develop a way to express a FIFO ordering. For instance, a FIFO waiting place could be invented and used for *Receive*. Although this might not be essential at a timethread level in order to keep the diagram simple, this problem definitely needs to be solved at the architecture level.

- This trace detects another problem related to the disconnection phase. Box 4 shows that the initiator SUD still receives information after the disconnection (*Disconnect* and *EndRec*) and even after a new *Contact* request. Box 5 indicates that the initiator SUD also plays this information. The latter can mislead the initiator who thinks he/she contacted someone else. Again, the designer learns that the system needs a mechanism to ensure that all information contained in buffers and queues is played (or thrown away) before the disconnection to be considered complete.

Simulation helps the designer to find such problems and to ask many questions. We must recall that this timethread map does not intend to solve all the problems issued from specification executions. The problems are noted and they will be solved later on, possibly in a new timethread map (if the solution is obvious in the timethread domain) or at the architecture level.

Testing

Appendix D.II presents three simple test cases, based on *Connection phase* use cases (fig. 69), that were applied to the *Telepresence* specification with the help of the tool LOLA.

The first acceptance test (spec. 13) and the first rejection test (spec. 14) helped in validating the output of the *Report* when the door is closed. No problem was detected.

The second acceptance test case (spec. 13), which was more complex, aimed at testing whether the *Report* output was always *Success* (or possibly *TimeOut*) when the responder *Open*s his/her door. This process was proved to be a `MAY PASS` test case. Although our test was generally executed successfully (99.94% of the finished executions), it was not always the case. The problem seems to be that when a responder *Open*s the door, the update of the internal state is not instantaneous and a *Knocking* request might have just the time to decide that the door is closed, denying the access as a result. The *OneExpand* command of LOLA helped in finding an unsuccessful execution revealing this race problem. *TestExpand* with the option `-s` could also be very useful to get all the unsuccessful executions. Testing the *Telepresence* specification can therefore be useful in finding problems associated to specific scenarios.

Knowing when to stop testing the specification becomes a legitimate question at this point. Usually, we can stop when there are no more bugs to discover, or when there is no money or time left [Pro 92]. More practially, we could define different software-reliability measurement [MuA 89] for timethreads. For instance we could test that:

- All valid use cases or scenarios can be executed;
- All invalid scenarios stated in the requirements are rejected;
- All timethread activities and paths can be traversed;
- All loops are traversed a certain number of times;
- All combinations of tag values & guards are executed;
- etc

We also might have to truncate to a certain depth the LTS corresponding to the LOTOS specification (when branches are infinitely long), and to execute a certain number of times the tests related to non-deterministic choices. To get confidence in our design, we therefore need different coverage criteria. We do not provide them in this thesis; we only present instances of use of testing. Coverage of timethread maps is still a research issue.

6.4 Transformations and Second Timethread Map

We are now at a stage where end-to-end timethreads have to be factored in order to get only one SUD that satisfies the requirements of both system roles (initiator and responder). The factoring procedure is explained, for the *Message Transfer Unit* example, in [Buh 93] and [BuC93]. In this section, we will not repeat everything that has been said about the whole method. We concentrate instead on the transformation phase, on some points in the mapping phase, and finally on the validation phase.

6.4.1 Transformation Phase

Getting one SUD that includes both roles (initiator and responder) from the timethread map of figure 75 might seem a very complex task. However, timethreads allow transformations that ease this factoring. The procedure can be explained in three steps, which are illustrated for a simple timethread in figure 80.

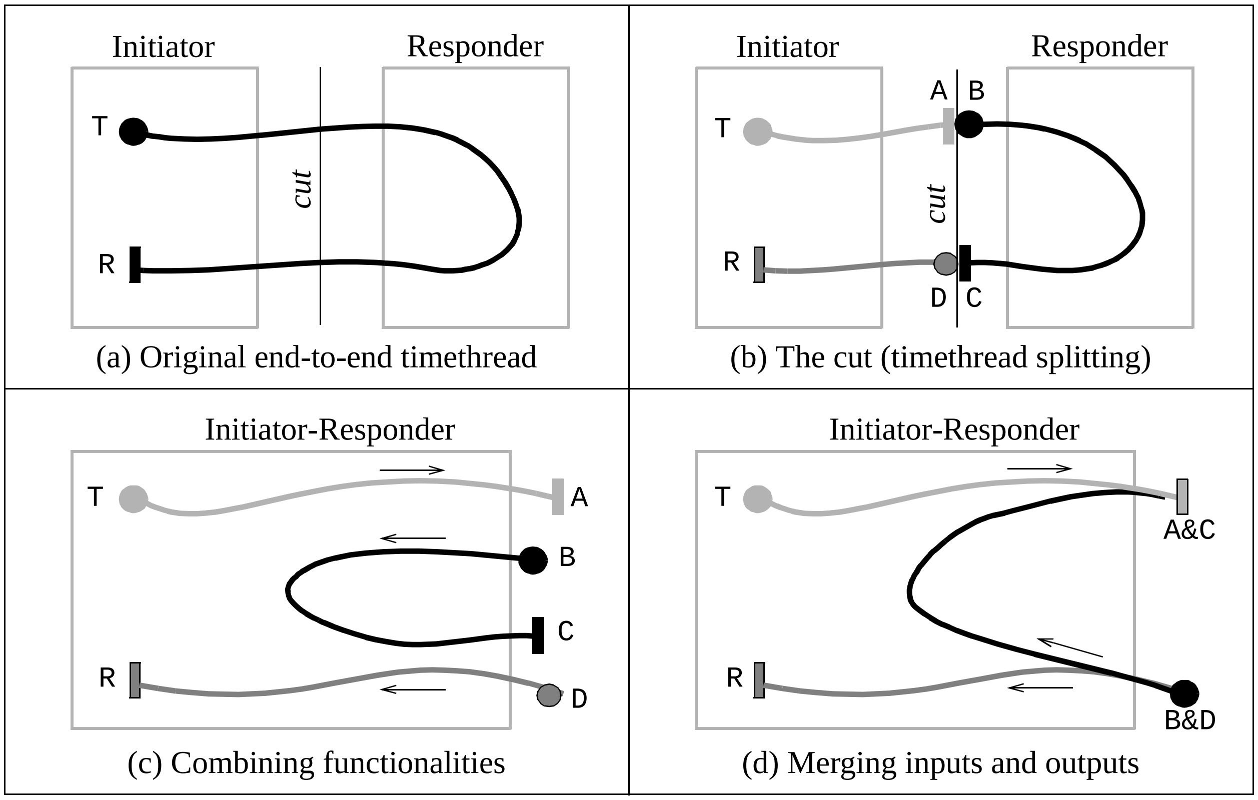

Initiator
Responder
T
R
cut
(a) Original end-to-end timethread
Initiator
Responder
A
B
T
cut
R
D
C
(b) The cut (timethread splitting)
Initiator-Responder
T
A
B
C
R
D
(c) Combining functionalities
Initiator-Responder
T
A&C
R
B&D
(d) Merging inputs and outputs

Figure 80: Factoring procedure

Factoring Procedure

Assume we start with a simple end-to-end timethread triggered by `T` on the initiator side, causing some activities (not shown here) in the responder's system, and resulting into `R` back in the initiator's system. Figure 80(a) presents such a timethread, which is very similar to timethreads *Reception* and *Knocking* from the *Traveler System*.

- Because the timethread spans over two systems (with complementing roles), we can assume that these systems communicate with each other to implement the causality flow. Hence, we can cut the end-to-end timethread to make explicit the communication of the two systems. In figure 80(b), this cut results in three interacting timethreads, which are assumed to be path equivalent to the original one (see the discussion on path equivalence in §5.3.2).

- Now that we have enhanced communication functionalities between the system, we must combine the functionalities of both systems into one initiator-responder system. To do so, we mirror the middle timethread (the one in the responder system) and bring it into the new complete system (fig. 80(c)). We added arrows to express the causality flow more clearly. At this point, the factoring is completed.

- Although our goal might be considered achieved in figure 80(c), we may wish to have only one input channel and one output channel on which the messages are multiplexed. In this case, we have to merge inputs together and merge outputs together to get a more realistic model. We did so in figure 80(d), and we obtained a system that possesses functionalities of both initiator and responder roles, with only two uni-directional channels. This last step is sometimes not needed as it might lead to overspecification.

Communication

Such complete SUD can be easily connected to another similar SUD. For instance, figure 81 shows two communicating systems. The output of the first system interacts with the input of the second system, and the input of the first system interacts with the output of the second system.

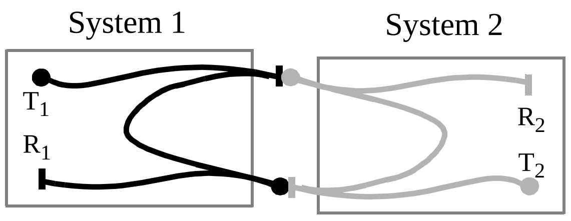


Figure 81: Two interacting complete systems

The interactions can also occur via an underlying service or medium (gray box in fig. 81). The latter is placed between the two systems and could manage the routing, the retransmissions, etc. Two complete *Telepresence* systems could communicate, for example, via an underlying service provider that can transport data (voice, video, requests, responses...) over some medium in a reliable way. For instance, this could be a XTP protocol over a high-speed ATM optic fiber network.

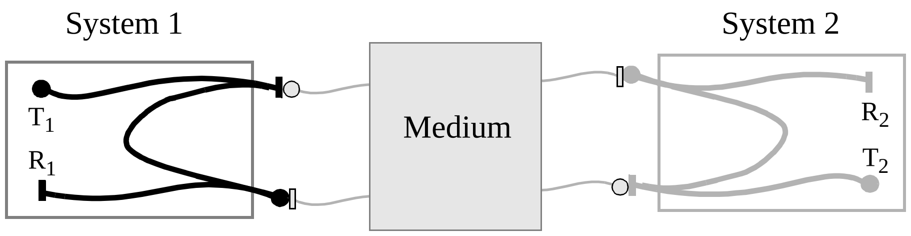


Figure 82: Two complete systems interacting via a medium

Allowing these types of communication may however lead to causality flow problems. In figure 81, when a token goes from system 1 to system 2, it has a non-deterministic choice between two alternatives. One of them, i.e., where the token goes directly from T_1 to the end result R_2, was not allowed in the original system. This problem has to be taken care of with tags and guards. By adding tags indicating the path of provenance before going out of a system, it is possible to resolve this non-determinism. We also need guards on each alternative when entering a system, so that we can use tag values to guide the tokens to their right path. This mechanism is used in the second *Telepresence* map (fig. 84).

Telepresence Factoring

The factoring transformation is now about to be applied to the first *Telepresence* map. The vertical cut will occur between the initiator and the responder. If we take the map of figure 75 as it is, we create five new timethreads after the cut. However, this number can be reduced to four if we apply a transformation on timethread *Reception* before the cut. Figure 83 illustrates this transformation:

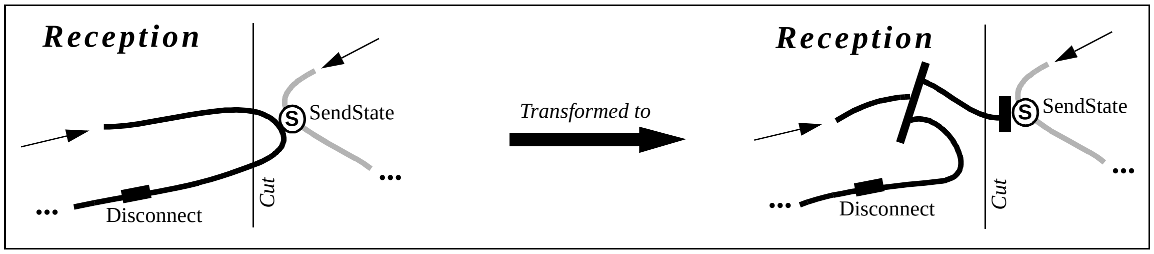


Figure 83: Transformation of Async in Reception

The asynchronous event `SendState` is transformed into *Reception*. We extract a parallel path resulting in `SendState` and another parallel path which is the continuation of the original *Reception* timethread. This path equivalence relation has the advantage of reducing the communication coupling between systems after the cut.

This transformation having been applied, we now execute the factoring procedure by cutting the timethreads and combining the functionalities (as in fig. 80). We assume here that we want to explicitely show that there are two unidirectional channels per system only. Therefore, we use the last step (d) defined in the factoring procedure, and we merge inputs and outputs. The second timethread map is presented in the next figure:

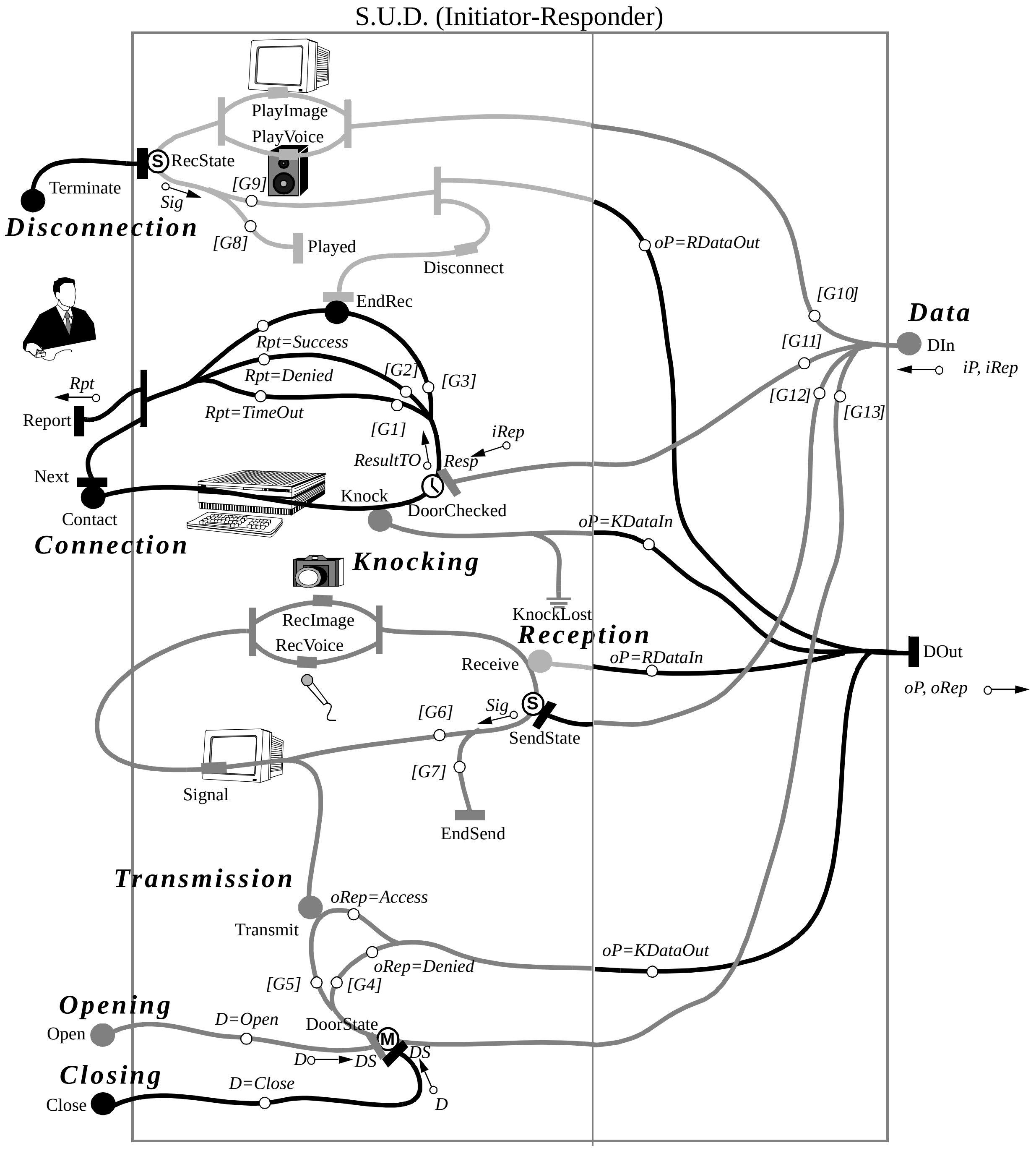


Figure 84: Second Telepresence timethread map (1SUD)

Here are some comments on figure 84:

- Timethreads *Disconnection*, *Connection*, *Transmission*, *Opening*, and *Closing* are kept unchanged because they were not affected by the cut.
- The dotted vertical line shows were the cut was done.

- The merging of inputs results in a new and complex timethread called *Data*. It includes some paths and functionalities from the original *Reception* and *Knocking* timethreads.
- Two new external events are created for communication (`DIn` and `DOut`). They also take care of the tags flow.
- As discussed previously in this section, tags and guards have been added to resolve non-determinism associated to communication. The tag `oP` (standing for output path) indicates the last path taken and is sent to a communicating system via the result `DOut`. The other system receives this information as `iP` (input path). This tag is then used in the new guards `G10` to `G13` to route tokens correctly. These guards are:

  `[G10]` is `[iP eq RDataIn]`
  `[G11]` is `[iP eq KDataOut]`
  `[G12]` is `[iP eq RDataOut]`
  `[G13]` is `[iP eq KDataIn]`

This complex transformation leads us to a map representing a single *Telepresence* SUD that has functionalities of both roles (initiator and responder) defined in the requirements.

Note that the identity of some original timethreads (*Knocking* and *Reception*) appears to be lost in the factored map. We can see that tags and guards in timethread *Data* intend to preserve the possible causality paths in the system, but somehow we lose some structural information identified in the first map. Mechanisms to ensure we can still access this information would be welcome here. This is yet another research topic to be addressed in future work.

6.4.2 Mapping Phase

We apply now the interpretation method in order to get our second LOTOS specification of the *Telepresence* system. We first apply the decomposition to get the STDL and LARG descriptions, then we binary group the latter using the LAEG method. Finally, the LOTOS specification is generated from these descriptions.

STDL Descriptions

Since timethreads *Disconnection*, *Connection*, *Transmission*, *Opening*, and *Closing* have not been affected by the factoring procedure, their STDL descriptions remain unchanged (refer to fig. 85). In figure 85, we present the descriptions of timethreads *Data*, *Knocking*, and *Reception*.

```
Timethread Data is
   Internal PlayImage, PlayVoice, Disconnect
   Trigger (DIn ? iP ? iRep)
   OrFork
         Guard ( not((iP eq KDataOut) or (iP eq RDataOut)) ) (* [Gadded] *) Continue
      Or
         Guard (iP eq KDataOut) (* [G11] *)       (* End of Knocking *)
         Path Result (DoorChecked ! iRep) EndPath
      Or
         Guard (iP eq RDataOut) (* [G12] *)       (* End of Reception *)
         Path Result (SendState) EndPath
   EndOrFork
   Choice
         Guard (iP eq RDataIn) (* [G10] *)        (* Body of Reception *)
         Path
            Par
                  Action (PlayImage)
               And
                  Action (PlayVoice)
            EndPar
            Wait Signal (RecState)
            OrFork
                  Guard (Sig eq Yes) (* [G9] *) Continue
               Or
                  Guard (Sig eq No)  (* [G8] *)
                  Path Result (Played) EndPath
            EndOrFork
            AndFork
                  Path
                     Action (Disconnect)
                     Result (EndRec)
                  EndPath
            EndAndFork
            Tag (oP = RDataOut)
         EndPath
      Or
         Guard (iP eq KDataIn) (* [G13] *)        (* Body of Knocking *)
         Path
            Wait Memory (DoorState ? DS)
            Choice
                  Guard (DS eq Close) (* [G4] *)
                  Tag (oRep = Denied)
               Or
                  Guard (DS eq Open) (* [G5] *)
                  Async (Transmit)
                  Tag (oRep = Access)
            EndChoice
         EndPath
   EndChoice
   Result(DOut ! oP ! oRep)
EndTT
```

```
Timethread Knocking is
   Trigger (Knock)
   Loss (KnockLost)
   Tag (oP = KDataIn)
   Result (DOut ! oP)
EndTT
```

```
Timethread Reception is
   Trigger (Receive)
   Tag (oP = RDataIn)
   Result (DOut ! oP)
EndTT
```

Figure 85: STDLdescriptions of the second Telepresence map

The STDL grammar works well when used in a constructive approach, i.e, no problem occurs when we build timethreads from simple ones to more complex ones. However, when transforming a timethread, we may observe a limitation of the grammar w.r.t. mixing choices and or-forks. When the timethread *Data* is triggered (`DIn`), the token has to take one out of the four paths in front of it. This is why guards have been placed. However, what is implicit here is an or-fork with three possible paths: one guarded with *`[G11]`*, another guarded with *`[G12]`*, and the last one (unguarded) which leads to the continuation of the timethread. This last path then has a choice between two sub-paths, guarded with *`[G10]`* and *`[G13]`*. Now, the problem is that a deadlock may occur at this point. When a token has `ip=KDataOut` as information, it may non-deterministically follow the ungarded path of the or-fork, and then deadlock in front of the choice.

A simple solution to this problem is presented in figure 83. By adding a guard to the third option of the or-fork, we can route the tokens correctly. The guard (*`[Gadded]`* in the right timethread) must forbid access to its path to tokens that are allowed to take one of the other paths. To do so, such guard must have the following format: **`not`**`((first guard)` **`Or`** `(second guard)` **`Or`**`...)`. In our case, we have:

`[Gadded]` is `[not( (iP eq KDataOut) or (iP eq RDataOut) )]`

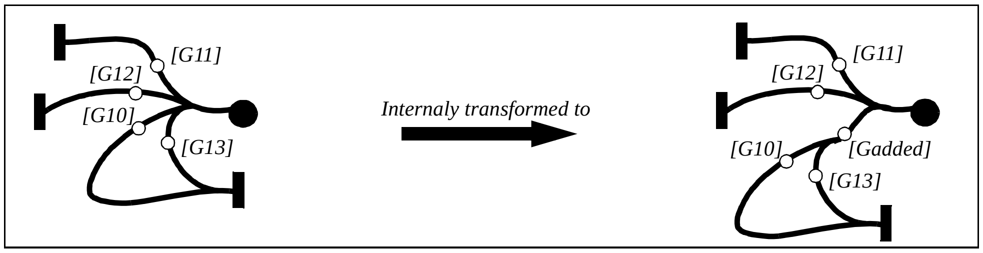


Figure 86: Internal transformation

This solution was adopted in the STDL description of *Data*. If a tool was to automate the factoring transformation, this feature would have to be included.

LARG Description

The *Telepresence1SUD* LARG containing our eight interacting timethreads is shown in figure 77:

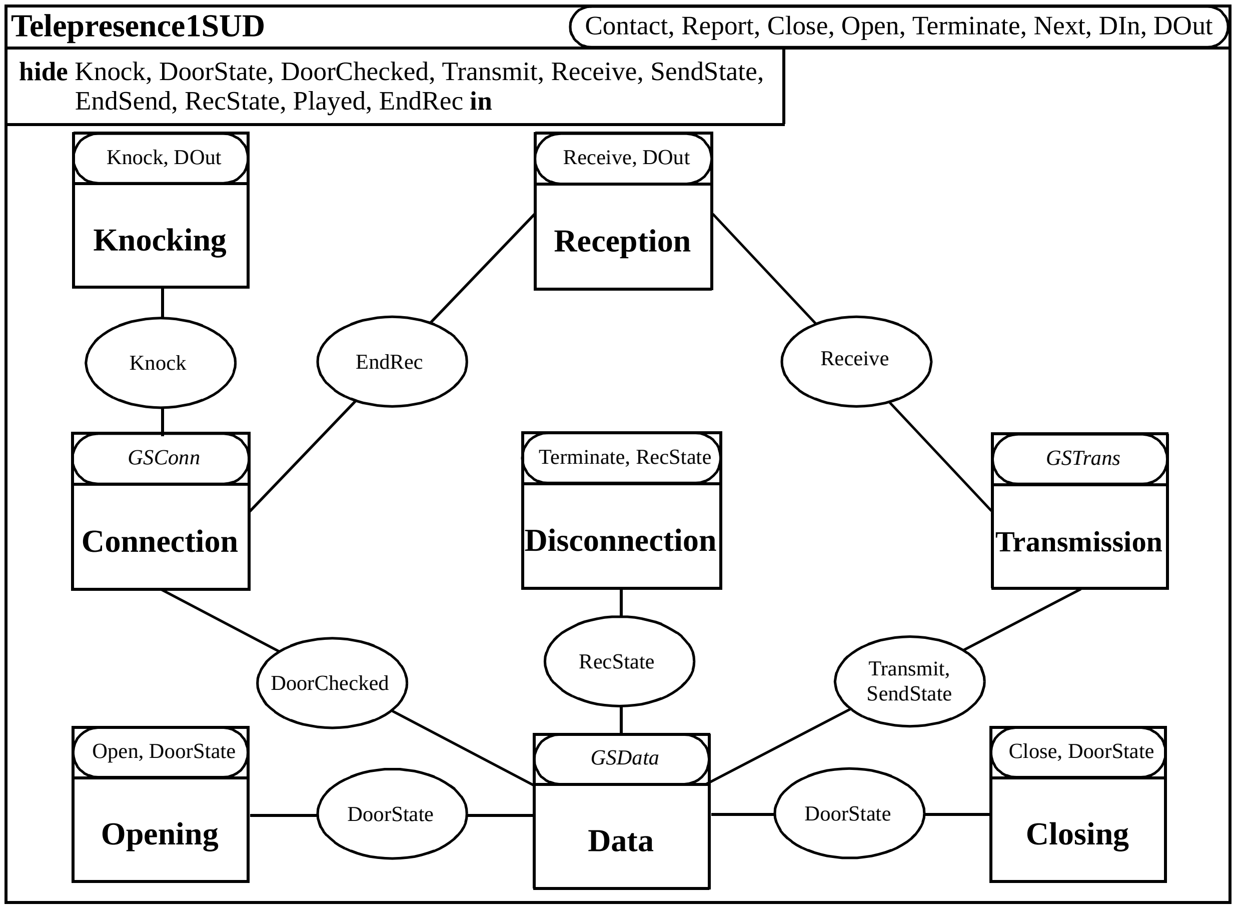


Figure 87: LARG of the second Telepresence map

We enumerate here the interfaces (italicized label sets) which were too lengthy for our diagram:

- *GSConn* is `Contact, Knock, Doorchecked, EndRec, Report, Next`
- *GSData* is `DIn, DOut, SendState, DoorChecked, DoorState, Transmit, RecState, Played, EndRec`
- *GSTrans* is `Transmit, Receive, SendState, EndSend`

Binary Grouped LARG

With the help of the LAEG method, we generate the binary grouped LARG (fig. 78) from the ungrouped one.

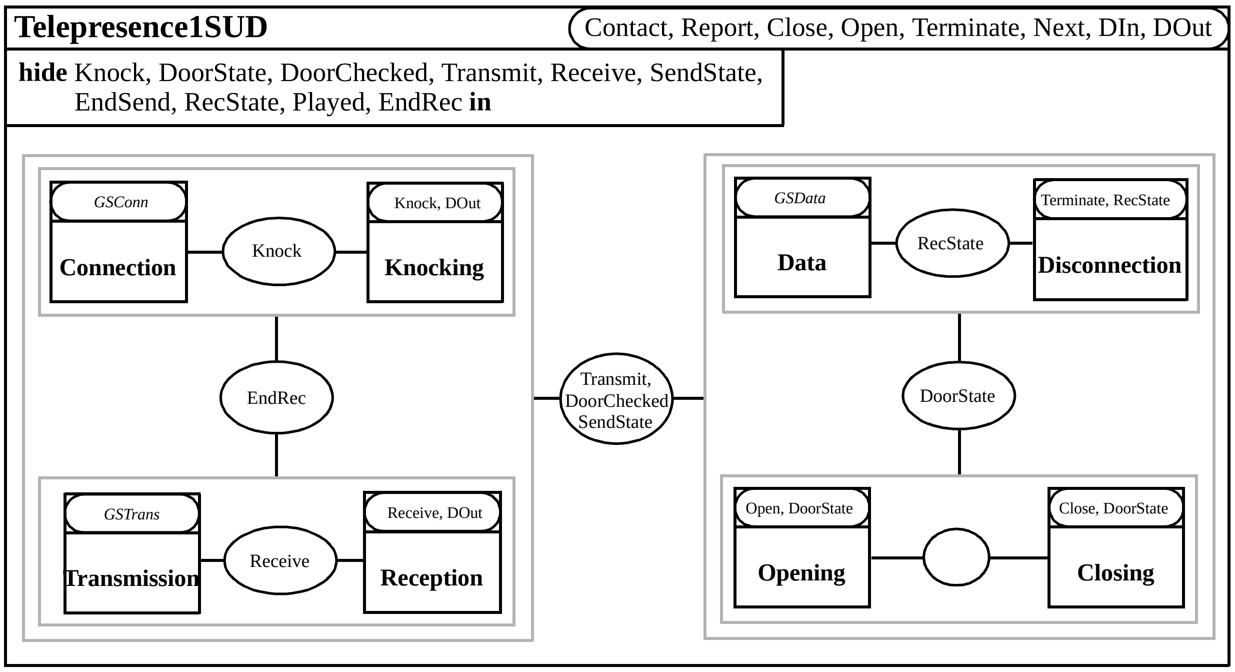


Figure 88: Binary grouped LARG of the second Telepresence map

As usual, the interfaces and hidden activities from the ungrouped LARG are kept unchanged in the binary grouped LARG.

Mapping on LOTOS

As suggested by the method, the structure part of the specification (lines 52 to 81 of spec. 16) is generated from the binary grouped LARG. The processes corresponding to the individual timethread are (manually) derived from the STDL descriptions. Appendix C (spec. 16) presents the complete LOTOS specification named *Telepresence_1SUD*. We consider that a level 1 specification with recursion is sufficient for validation purposes in this thesis. Therefore, the level 3 specification will not be given nor used.

6.4.3 Validation Phase

New Specification

Telepresence_1SUD (spec. 16) contains the functionalities of the initiator and the responder roles. The validation of such a specification is not obvious at first sight, since end-to-end behaviours are not present in the same way as in the first specification. Each time an initiator sends data to a responder (or a responder to an initiator), the resulting event is *DOut* with some associated tag values. Each time a responder or an initiator receives data, the external event is *DIn*. The specification shows the correct paths, but many forbidden scenarios could occur without proper constraints from the environment. For instance, an initiator could receive voice and video data even if he/she did not send any contact request.

To keep this fact in mind when we play this specification is a tedious task. One natural way of constraining this specification correctly is by composing the latter with a similar system (§6.4.1). Therefore, we will not use the specification *Telepresence_1SUD* as is, but we will transform it into a process as part of a new specification corresponding to the communicating systems of figure 81. Two process instances will be synchronized on their communication channels (`Din` and `DOut`).

We can easily create such specification (called *Telepresence_2Systems* in spec. 10). We double the gates in the global interface in order for each systems (1 and 2) to have their own complete interfaces. Then, we synchronize two instances of the process *Telepresence_1SUD* on the channels `In1Out2` and `In2Out1`[1]. These are all the modifications needed to get the communicating systems we wanted. Of course, we could specify a medium in between (as in fig. 81). This would allow us to test more closely the robustness of the system. However, since we do not want the model to become too complex, we will stick to the first option.

1. In this example, we choose to hide these channels from users

```
specification Telepresence_2Systems [ Contact1, Contact2,
                                      Report1,  Report2,
                                      Close1,  Close2,
                                      Open1, Open2,
                                      Terminate1, Terminate2,
                                      Next1, Next2] : noexit


... (* Definition of libraries and the Tag ADT *)


behaviour (* Composition of the 2 similar systems *)

hide In1Out2, In2Out1 in

      Telepresence_1SUD [Contact1, Report1, Close1, Open1, Terminate1, Next1,
                         In1Out2, In2Out1]
   |[In1Out2, In2Out1]|
      Telepresence_1SUD [Contact2, Report2, Close2, Open2, Terminate2, Next2,
                         In2Out1, In1Out2]

where

   process Telepresence_1SUD [ Contact,  (* Initiator want to contact responder *)
                               Report,   (* Result of a contact *)
                               Close,    (* Close user’s door *)
                               Open,     (* Open user’s door *)
                               Terminate,(* Initiator terminates a connection *)
                               Next,     (* Get ready for next connection *)
                               DIn, DOut (* Incoming/Outgoing data*) ] : noexit :=

   ...  (* LOTOS code from the original specification Telepresence_1SUD *)

   endproc (* Telepresence_1SUD *)

endspec (* Telepresence_2Systems *)
```

Specification 10: Telepresence (2 systems composed together)

Validation Strategy

We need to validate *Telepresence_2Systems* against the requirements and the previous specification (*Telepresence*) in order to see that:

- It conforms[1] to the specification *Telepresence* (which was itself assumed to conform to the requirements). The conformance checking is due to the presumed “path equivalence” between the two specifications, resulting from the factoring transformation.
- It does not create more problems than there were in *Telepresence*.
- It solves some problems found in *Telepresence.*

1. As explained in section 5.4.2, we still use design testing to check conformance between specifications and not conformance testing

Hence, we can re-apply the simulation sequences and test cases used in section 6.3.5. New tests cases can be derived from the *Telepresence* specification. Also, specific test cases validating the factoring procedure and its communication aspects can be generated.

In the validation examples, we will use testing only. To simplify the approach, simulation and verification are not used here, although they would be necessary in a real-life validation. We reuse the three test cases (spec. 13: *AcceptTest1*, spec. 13: *AcceptTest2*, and spec. 18: *RejectTest1*) previously defined in appendix D.II. Of course, they are adapted to match the new gate names (`Contact1`, `Result2`...).

We also add another rejection test called *RejectTest2* (spec. 20). This last test was not in the previous test suite. We create it here in order to ensure that we have no additional error related to invalid scenarios due to communication between systems.

Telepresence_2Systems does not intend to solve the problems we detected earlier in *Telepresence* (§6.3.5). The former is the result of a factoring transformation that preserves path equivalence. Therefore, we will not create additional test cases for these problems, which obviously will have persisted in the second specification.

Many other test cases could be generated for design testing. However, to simplify the results and to ease comparisons, we will consider the four tests presented only.

<u>Testing Results</u>

Appendix F presents the results of our four test cases. Here are some general conclusions from this experiment:

- *Telepresence_2Systems* has more internal events (due to `DIn` and `DOut`) than *Telepresence*, leading to an increased number of possible states and transitions.

- *AcceptTest2* is again a **`MAY PASS`** test case. This time, 99.92% of the finished executions terminated successfully. This ratio compares to the 99.94% found in section 6.3.5. Note that we could use simulation or the LOLA command *OneExpand*, or the `-s` option of *TestExpand*, to reveal the same race problem with the knocking request which was already detected in *Telepresence*.

- All three common test cases (*AcceptTest1*, *AcceptTest2*, and *RejectTest1*) performed similarly. They did not detect any major difference between the original specification and the transformed one. Therefore, they failed in proving that *Telepresence_2Systems* does not conform to *Telepresence*.

- The last rejection test case (*RejectTest2*) checked that a *Contact* request does not directly lead to a *Result* (with any parameter) on the other user’s side. Such process tests that the guards we added correctly route tokens, according to their prior paths to which tags were associated. The result showed that no problem was detected with this scenario.

This validation, although very superficial, did not detect unknown problems with this second specification of the *Telepresence* system. Hence, we might have a certain degree of confidence in that *Telepresence_2Systems* conforms to *Telepresence*.

CHAPTER 7 Discussion

This chapter presents several topics that go beyond the thesis objectives. However, we believe the following discussion to be useful by placing the thesis achievements in a wider context and by opening new horizons. The next sections are closely related to many issues developed in the previous chapters. We chose to emphasize four main topics which are: the architecture, the STDL grammar, validation in general, and a few ideas on possible tools.

7.1 Towards a First Architecture

Why would we need another design methodology using formal methods while several already exist? For instance, the Lotosphere Methodology [LOT 92], based on the conventional stepwise refinement, offers powerful structuring and abstraction facilities that allow designers 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 the mathematical foundations of LOTOS, that allow verification of properties and extensive support for simulation and testing.

We believe that 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 themselves. The Lotosphere Methodology takes a formal method (LOTOS) and tries to build a complete design methodology on it. In our approach, we start from some visual design concepts, very natural to designers. Then, we use formal methods to help formalizing parts of the visual notation with what formal methods offer the best. This concept was called *interpretation methods* (§3.1) in [Bor 93]. This research direction seems, from our viewpoint, to be very promising.

Although we know from this thesis that we can capture the main requirements with a timethread map and then get the corresponding LOTOS specification, this does not mean that we have a complete design methodology. We at least need a more complete implementation-oriented model, such as an architectural specification, of our timethread-designed system. Obtaining an architectural design consistent with a timethread design is however a complex task. Two major approaches are distinguished for this purpose: *derivation* (fig. 89a) and *validation* (fig. 89b).

Derivation: In this approach, an architectural specification of the system (***AD***) is obtained from the LOTOS mapping of the timethread design (***TD***) on which some correctness preserving transformations are applied. This new specification is mapped to an architectural design.

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

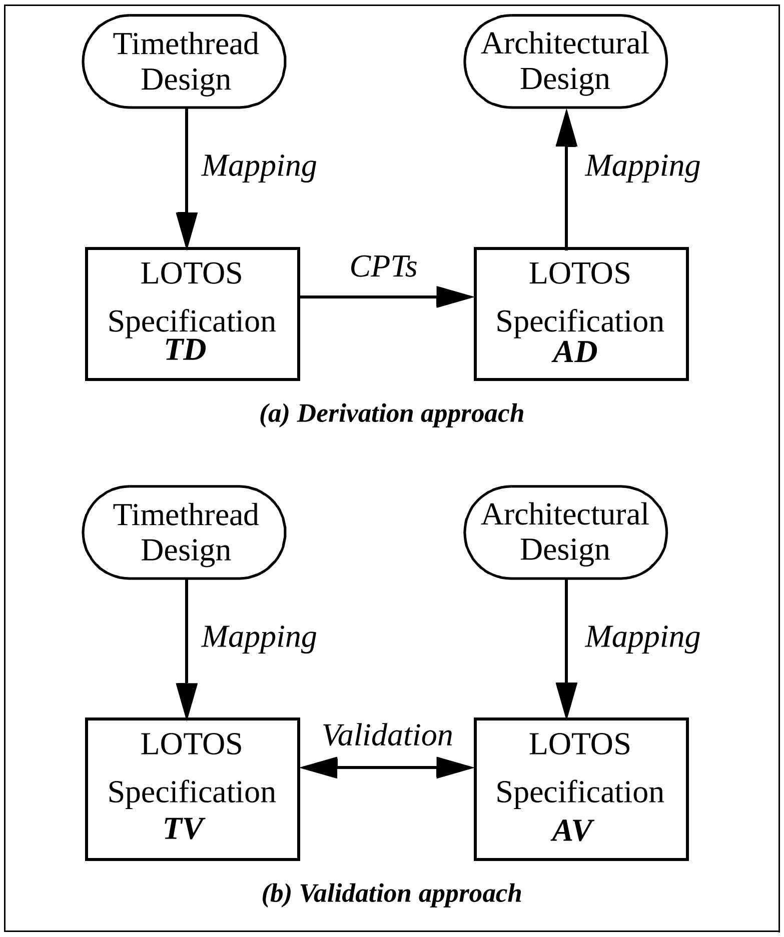


Figure 89: Derivation and validation approaches

The advantages of using CPTs in the derivation approach is quite obvious: no need for costly validation or for the generation of test cases. Although this solution is theoretically appealing, CPTs are often difficult to define and also hard to apply. Also, the derivation approach necessitates a reverse-mapping going from LOTOS to a particular architecture model, and we do not have any such method yet.

In [BuC 94b], the authors discussed an option we could relate to our validation approach (introduced in section 5.1.1). They explain how a timethread map allows the designer to look at several architecture options, leading to a collaboration graph (see the definition on next page) where roles and communications are more closely defined. We reproduced, in figure 90, the steps going from a timethread map ♣, to timethread-role map ➻, to the collaboration graph ➩ (the first architecture) previously presented in figure 53.

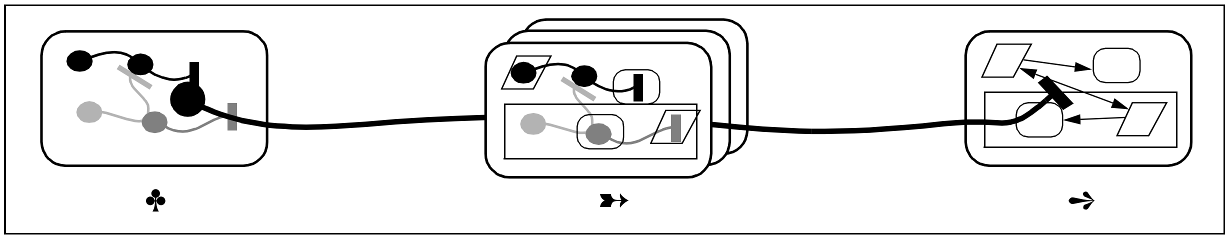


Figure 90: From timethread maps to collaboration graphs

A *role organization* is usually composed of simple role components (called carriers, workers, and teams by the authors) to which timethread activities are associated. Carriers carry whole segments of timethread forward, while taking care of concurrency issues. Workers do the functional work for carriers. We can think of workers as having point responsibilities along timethreads. Teams express collaboration grouping.

A *timethread-role map* is a timethread map on which we superimpose a role organization. We can compare different organizations of configurations w.r.t. the same timethread map, allowing designers to consider robustness, cost, and performance issues. Once the designer finds a configuration that satisfies his/her major criterion, he/she tries to derive a so-called *collaboration graph*.

While timethread maps show causality flow paths, collaboration graphs show control flow paths. They also aim at solving concurrency issues. Transforming timethread-role maps into collaboration graphs leads to potentially many solutions, so human judgement, commitments, and design decisions are required to determine the best one. This type of graph is considered to be a first high-level architecture, based on roles instead of concrete hardware or software components.

Some heuristics and guiding rules to help in getting a collaboration graph from a timethread map are presented in [BuC 94b], but no automated transformation procedure exists yet. This is why we can relate this approach to *validation*. By defining an interpretation model for a collaboration graph in order to get a corresponding LOTOS specification, we could validate the latter against the specification generated from a timethread map. What is interesting here is that collaboration graphs and timethread maps are not independent. Because of a certain degree of dependance (heuristics and guiding rules), we can informally derive a collaboration graph from a timethread map and then perform a formal verification according to some equivalence or conformance criterion.

Of course, we acknowledge that much work remains to be done in this area. Nevertheless, we believe that this approach, which combines many advantages of formality and informality, can lead (especially if automated) to better and more rapidly obtained RTD system designs.

7.2 STDL

7.2.1 Features Evaluation

A context-free grammar is often used to give a formal syntax specification of a language. We used an EBNF grammar (§4.2.3) to define in an abstract way the allowed paths of a single timethread. The grammar rules forbid many types of incorrect timethreads, without the help of additional static or dynamic semantic rules.

STDL considers single timethreads as entities in their own right. It provides readable descriptions that will ease the design of compilation tools.

Modifications can be brought to the language in order to expand or adapt it. This is also eased by the three grammar rules `<Seg>`, `<GenOptions>`, and `<WPOptions>`.

This language is well-suited for a constructive approach, i.e., starting from a simple timethread, the designer can make it more complex by adding segments. STDL also possesses many constructors that can be straightforwardly mapped onto LOTOS operators, and many levels of specifications can also be generated from the same STDL description. The tag mechanism is also a powerful feature that allow a more complete management of unfeasible paths, especially after transformations such as factorings (§6.4.1).

STDL, as it is now, is a major step towards the creation of a compiler tool that will generate formal (LOTOS) specifications from timethread maps. We think that STDL is a language suitable for LOTOS and a whole family of languages with multi-way rendezvous. In fact, we always had in mind a multi-formalism context while developing STDL. Therefore, because it is a general language describing timethreads for what they are, we believe other formal languages might be used as output of other interpretation methods, perhaps at the cost of minor modifications to STDL.

7.2.2 Current Limitations

STDL is neither a perfect language, nor the ultimate solution to all our design problems. If we want this language to improve with time, we first have to be honest and acknowledge its current limitations and those of the LOTOS interpretation method. Hence, we decided to point out in this section some of these limitations:

- STDL is an incomplete timethread description language because it does not manage timethread interactions; we had to use LARGs as a formal means to express them. A better language that would include both STDL and LARGs, in a consistent way, is still needed.
- We do not have any formal proof that a STDL description (with the map LARG) and the generated LOTOS specification are representative of the corresponding timethread map. Timethreads semantics is informal and therefore, since we cannot go formally from informal to formal, the formal semantics we gave to timethreads is hard to prove correct, complete and consistent. We simply did our best in that matter, and perhaps it is hard to do otherwise!
- Because we do not have a complete algorithm (or a compiler) yet, the mapping process (from STDL to LOTOS) is not guaranteed. We saw that, with a level 1 specification, the mapping method seems straightforward and correct. Level 2 and 3 specifications are more complex and the mapping may proved to be much more difficult than it appeared for level 1.
- STDL was influenced by the limitations of our interaction model (LARGs). Since the LARGs only have synchronous interactions where no direction is specified, the grammar had to include these features, although they belong to the interaction domain. This led to the creation of the constructors **`Sync`** and **`Async`**, and the tag flow operators **`?`** and **`!`**.
- The STDL tag mechanism is strongly related to LOTOS Abstract Data Types (especially booleans and natural numbers) and the value offer operators `?` and `!`. Although this eases the mapping onto LOTOS, this coupling might be a problem when considering other formal languages as output to other interpretation methods.

7.2.3 Possible Enhancements

The current STDL is a context-free grammar that expresses the syntax rules of the language. Of course, as expressed in section 7.2.2, this is insufficient. We must define a set of semantic rules (static and dynamic) to complete the language. These rules combined with a set of complete mapping algorithms would formalize the language and the translation procedure.

STDL is a language that offers a lot of extension possibilities. We could extend it to include grammar rules to describe timethread interactions in a map. This new *Timethread Map Description Language* (or *TMDL*) would require constructors representing interactions, global hide operator for systems internal events, recursive groupings (needed for binary groupings), static semantic rules indicating valid and invalid interactions, etc. We believe SDTL can be reused as is, to describe single timethreads.

We mentioned (§7.2.1) ways to extend the STDL grammar. New segments can be added in the `<Seg>` rule, when a new timethread construct is created. `<GenOptions>` allows the addition of new global options to a single timethread. Finally, `<WPOptions>` allows different options for a waiting place. Of course, other grammar rules could be defined or modified to adapt new features or to suit a new target formal language. We could, for instance, add to junction points several options similar to the ones defined in `<WPOptions>`.

To give a more precise idea of an extension of the `<WPOptions>` rule, we defined two new types of waiting place, called *Signal* and *Memory*, in the *Telepresence* system (§6.3.2 and 6.3.3). This transformed STDL in the following way:

```
<WPOptions> = [<Delayed> | <Timed> | <Signal> | <Memory>]
<Signal> = Signal
<Memory> = Memory
```

We also gave, in those sections, the LOTOS interpretation of such waiting places. Although we want to express causality paths instead of behaviour at the timethread abstraction level, we saw that a few *enhanced waiting places* were very useful to play the design. This approach led to a much more realistic description of the *Telepresence* system and a better management of its causality paths.

By adding simple options corresponding to LOTOS internal waiting place machinery, we could simulate many types of waiting places:

- Normal (multiway rendezvous we have by default)
- Delayed (defined in the grammar)
- Time (defined in the grammar)
- Signal (defined in the *Telepresence* system)
- Memory (defined in the *Telepresence* system)
- Bounded buffer (unordered)
- FIFO or LIFO queues
- Priority queues (we could create *priority tags*)
- One-to-many, many-to-one, and many-to-many interactions. These types could be used, for instance, in the *Traveler* system where a cab could wait for 4 travelers (maximum) or a timeout before leaving. Then a plane could wait for 200 people before leaving, etc. These waiting places could be very useful for a mapping onto Petri nets, which supports many-to-many interactions in an elegant way.

Most of these complex waiting places could be useful at late stages in the timethread development process, before the mapping to an architecture where buffers, mailboxes, queues, etc., are de facto needed for communication management.

Inventing a simple and clear notation to distinguish the above cases and possibly many other ones is however not a trivial thing. We suggested the use of letters inside waiting places to show signal and memory waiting places, but some people may find this solution too cumbersome. There always is a trade-off between capturing every detail and making the big picture clear. Too much notation gets in the way of the latter, but again, a tool might hide such precise details at an abstract level and show them at the user's request.

7.3 Validation

Perhaps one of the greatest strength resulting from the interpretation method concept is that the validation allows a designer to use many existing tools to simulate, test, and verify different criteria, requirements, situations, scenarios, etc, in a timethread design.

We know that validation is necessary, even with incomplete descriptions, through any design process. A design should always be validated in some way (simulation, test cases...) against previous designs, and ultimately against the requirements. Good validation tools are of course needed at all stages.

We have discussed validation issues all along the design of the *Traveler* system and the *Telepresence* system. Although timethreads were not created to become a simulation model, we showed that timethread-based validation can help playing a design and point out different categories of problems. An *error model* regroups these categories for a given notation.

Since we used LOTOS as the formal underlying model for timethread maps, our error model helps designers to detect problems such as:

- Bad ordering of events,
- Unfeasible or unwanted synchronizations,
- Unfeasible or unwanted paths,
- Race conditions between tokens,
- Absence of path equivalence and/or conformance w.r.t. other specifications or requirements,
- Incorrect communication between systems, etc...

As we saw from the many examples we gave in the thesis, LOTOS offers many facilities and useful tools to detect these categories of problems. In fact, we can use all well-known LOTOS validation techniques to simulate, test, and verify a timethread-oriented specification. The only difference with respect to traditional LOTOS validation is that we play with possible paths instead of possible behaviour. Thus, the techniques have to be slightly adapted to accommodate this way of thinking.

Of course, the use of other formalisms (we already mentioned Petri nets, event structures, Z...) as target outputs of interpretation methods would allow other fault models. For instance, performance and time constraints can hardly be validated with LOTOS. A Petri net (with some extension) specification of a timethread map might be a better choice for these categories of problems. In fact, we are not the only ones believing that using more than one formalisms representing a system under design offers some advantages. In [BoC 93], another multi-formalism approach is discussed. The authors propose to build a meta-language for design, based on visual notations, that would use the specification and validation strengths of different formalisms. Although they use different techniques and ideas, this interest in multi-formalism makes us think we might be going in a useful research direction.

7.4 Tool Support

Timethreads and interpretations methods are more useful if tools can support them. In this section, we speculate on what is required and expected from such tools.

7.4.1 Internal Representation of Timethreads

Formal, Visual, and Internal Representations

In this thesis, we defined part of the *formal representation* of timethreads. We explained in section 7.2 that STDL could be extended with grammar rules representing the LARG interactions, hence forming TMDL. If we add semantic rules to check valid interactions, consistency, missing or invalid information, etc., we get a better formal representation of a timethread map.

Timethreads are a visual notation, thus graphical and spatial information have to be recorded somewhere. Our formal representation, as it is now, does definitely not include such information. An additional *visual representation* of timethread maps, internal to tools, is therefore needed. Although this is outside of the scope of this thesis, we think it could be composed of some graph grammar, suitable for 2-D graphs description, and a set of data specifying visual details.

A timethread tool would need a general description, called *internal representation*, including both the formal representation and the visual representation. We show the structure of this internal representation in figure 91.

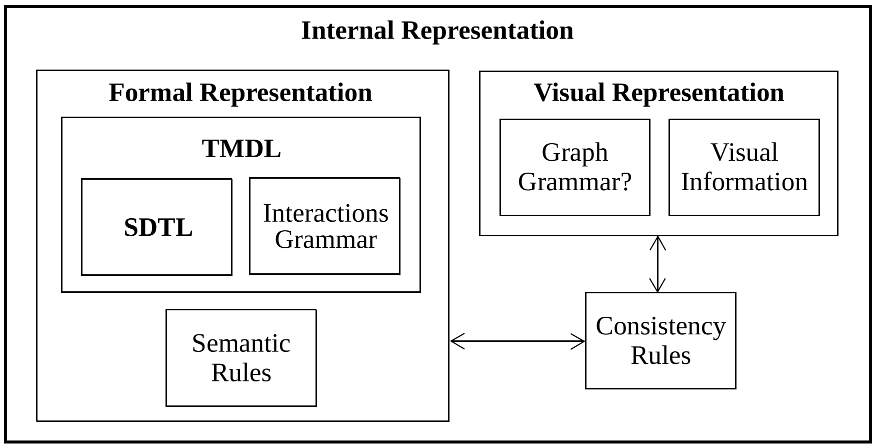


Figure 91: Internal representation of a timethread map

In order to maintain consistency between formal and visual representations, a tool might need a set of *consistency rules* as part of the internal representation.

This internal representation of timethreads maps will be especially useful for interactive design and for automated generation of formal methods specifications.

Construction and Recognition

We see two different ways of using the internal representation in a tool: a *recognition* approach and a *construction* approach.

In a recognition approach (fig. 92a), the designer works on the visual representation via a Graphical User Interface (GUI) only. When the timethread map is satisfactory, the tool uses a *recognition method* to generate the map formal description. This leads however to some major difficulties:

- The visual representation has to be complete and rich enough to provide all the information needed to construct the formal description.
- We need a recognition method to compute or evaluate a formal representation from a visual one instead of simple consistency rules. We believe the former to be more complex to define and implement than the latter.
- Since the recognition is done when the map is finished, the designer does not know if this map is valid or not since no semantic rules have been applied.

With this approach, we lose in a sense the need for a separate formal representation. Everything would have to be included in one model (the visual representation), and then the formal representation would only be a projection of this model.

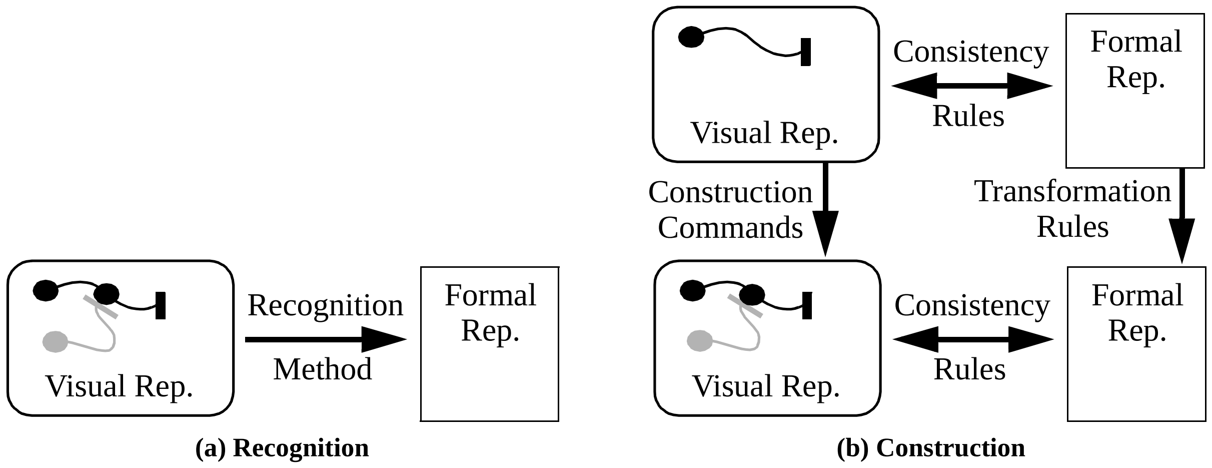


Figure 92: Recognition and construction approaches

The construction (fig. 92b) is slightly different. In this approach, the designer still works on the visual representation via the tool GUI, but at the same time, the formal representation is built and consistency is checked. The user applies construction commands and these are translated internally into transformation rules. Since the latter are validated by the semantics of the language, the mutual consistency of these various representations is assured. The complete map is thus created by construction, from a simple timethread to a complex map. In this way, the tool constrains the construction commands allowed to the designer, and only valid timethread maps can result.

Of course, we still are looking for better solutions, but this one seems promising and feasible. This is part of work to be done in other theses and projects.

<u>Interpretation Methods and Internal Representation</u>

The visual interpretation includes all the information that is not pertinent to interpretation methods. Therefore, formal specifications are generated from the formal description of timethread maps only. The same formal description can be used as input to different interpretation methods.

7.4.2 Timethread Editor

The purpose of an editor is to provide a GUI allowing the designer to construct and transform timethread maps. Using a construction approach, the GUI would mostly manage the visual representation while consistency rules would update the formal representation. We can implement many different features in such a tool:

- The input could be a text file, a mouse-driven interface, or a pen-based interface.
- The GUI would allow commands for timethread construction and transformation. The commands and operations allowed would be based on CPTs, extension or equivalence criterion, and semantic rules. Such operations could look like:
    - Add/remove a timethread, and action, an event,
    - Connect timethreads (synchronously/asynchronously),
    - Insert a loop, a choice, a stub...,
    - Hide actions, events...,
    - Parallelize actions,
    - Split/merge/factor timethreads...,
    - etc...
- Abstraction facilities (hiding, stubs, layering, magnifications...) could be implemented.
- The tool could provide interactive management of tags and guards to eliminate unfeasible paths.
- Options for timethreads and waiting places (as defined in STDL) could be used.
- Timethreads would have attributes such as different colors, patterns, shapes...
- Comments, annotations and pictures could be inserted.
- We could provide a library of stubs for reuse.

This list is not exhaustive. Many other features could be defined in an editor to help him/her express more intuitively and clearly causality paths in a system.

7.4.3 Interpretation Tools

An interpretation tool maps the formal description of a timethread map onto a formal language by using an interpretation method. We can program many such tools, which are generally compilers, but they all have the same input: a TMDL description.

We can also define different options for specific tools. For instance, a LOTOS interpretation tool could generate different levels of specifications. Each timethread in a TMDL description could be associated to a specific level (1, 2, or 3), with or without recursion.

Interpretation tools might have to do some internal processing of the TMDL description. For instance, interactions in a plain TMDL description have to be binary grouped (using the LAEG method) before LOTOS code can be output.

7.4.4 Validation Tools

These tools already exist. This is one of the most important benefits of interpreting timethreads. Different target formal languages naturally lead to different validation tools and techniques. We already saw many LOTOS-based techniques and tools for simulation, testing, and verification. These types of tools exist for most formal languages, and they could all be used to validate particular aspects of timethread designs.

7.4.5 Other tools

Optimistically, we can foresee other tools especially useful at later stages of the design process:

- Going towards an architecture, we could add role architectures to timethread maps, leading to more realistic architectural specifications.
- We could have “intelligent” tools suggesting candidate architectural solutions by using different heuristics.
- We could design tools that, based on experience and specific architectural criterion, would propose different skeletons of collaboration graphs that fit a role architecture and a timethread map. Control and data could be inserted at this point.

- We could add animation to tools, in order to show problematic scenarios detected using validation tools.
- Performance analysis might be useful to designers, especially when considering different candidate architectures. A tool to analyze performance according to designer criterion would be very welcome.
- We could even think, for the last stages of the design process, about partially automated code generation.

As one can see, there is still much room left for hard work and imagination.

CHAPTER 8 Conclusion and Future Work

8.1 Conclusion

8.1.1 Objectives and Requirements Achieved

This thesis presented a LOTOS interpretation method for timethreads. We demonstrated that it is possible to generate meaningful LOTOS specifications, from timethread maps (*O1* in chapter 1), that can be used to validate and play the design in the early stages of the life-cycle methodology (*O2*). We showed how tools can support this transformation (*O3*), and we discussed many resulting issues and difficulties (*O4*).

We believe our four main objectives (*O1* to *O4* in chapter 1) were achieved. We also considered the five requirements, also enumerated in chapter 1, in the following way:

R1) Timethreads are a description model suited for RTD systems. We assume designers do not really change their way of thinking and working by using them.

R2) Our method is based on a multi-formalism approach. Although we used LOTOS in this thesis, other target formalisms can be considered. We had in mind the generality of STDL to satisfy this requirement.

R3) If tools are supported, designers do not have to be LOTOS experts to use our method, although a minimal knowledge is always useful. They simply have to use already existing validation and verification tools that generally possess user-friendly interfaces.

R4) Major system functionalities and basic scenarios are captured via a visual notation (timethreads, in occurrence) easier to conceptualize than plain textual descriptions.

R5) Design tools are not yet available for our method, but major steps towards their creation have been taken. Validation and verification tools however already exist. This reuse is an important advantage of our method.

8.1.2 Contributions

Four major contributions of this thesis were introduced in section 3.2:

LOTOS Interpretation Method for Timethreads

Based on the concept of formal interpretation method, our approach generates LOTOS specifications from timethread maps (chapters 3 and 4). In this thesis, we completed the method by providing the mapping sub-method for single timethreads. This method was successfully applied to two case studies.

Timethread Grammar

We defined a Single Timethread Description Language (STDL) to represent single timethreads (chapter 4). We used a context-free grammar to express construction rules. This grammar formalizes parts of the Timethread notation by itself, without any major reference to target formal languages. Transformation and mapping rules were enumerated for the generation of LOTOS code from STDL descriptions. Static and dynamic semantic rules are still needed to complete the language.

Techniques

Chapter 5 presented several techniques in the context of part of a timethread life-cycle methodology. We applied the interpretation method (mapping techniques) to get a LOTOS specification from the *Traveler* system. Then, we introduced several LOTOS-based transformation techniques. We discussed notions of equivalence, extension, and conformance in a timethread context. Validation techniques were also described. We largely discussed simulation, testing, and verification techniques w.r.t existing LOTOS tools. These techniques help designers in their thinking and analysis process by providing different ways to play the design and therefore to discover potential problems early in the design process.

Case Studies

We developed two case studies in the thesis. The *Traveler* system was a case study used to illustrate the different steps of our approach. It provided a dynamic context easy to understand. The second case study, the *Telepresence* system (chapter 6), was a more complex real-life RTD system. The visual contact service was defined in terms of its functionalities and basic components. A first timethread map was constructed from basic use cases and mapped onto STDL and LARG descriptions. We then manually generated its LOTOS specification using our interpretation method, and then validated it against the requirements with simulation and testing. Some test cases and simulation scenarios led to the discovery of several problems. We used a transformation called factoring to get the second timethread map. We generated the corresponding STDL, LARGs, and LOTOS specification. We validated the latter against the requirements and the previous specification to check conformance.

Other contributions have been identified along the thesis:

- Our LOTOS specifications use what we called a *timethread-oriented style*, different from other traditional specification styles that usually describe a system’s complete behaviour. This new style leads to specific approaches for validation and conformance checking because one deals with causality paths instead of pure behaviour.
- We identified different complexity options, known as *levels of specification*, for the generation of LOTOS specifications from timethread maps.
- We defined a *tag mechanism* very useful for path control in a timethread map.

- The STDL grammar can be *extended* in many ways, thanks to some rules for options and segments definitions. We also identify possible *enhanced waiting places.* Two of them (`Memory` and `Signal`) were used in the *Telepresence* system.
- In chapter 7, we discussed ways of using our approach to get a first architecture consistent with its timethread map.
- Also in chapter 7, different ideas on a complete timethread-oriented design tool are introduced. We discussed internal representations and tools for timethread maps edition, interpretation, and validation.

8.2 Future Work

Objective *O4* was to present problems with the approach and resulting research issues. Many topics presented in this thesis require further attention. We can easily define short-term and long-term research issues. We believe the most important ones are the following:

Short-Term Research Issues

- We use two different models to represent our timethread maps (LARGs and STDL). This causes non-homogeneity problems. STDL could be extended to include timethread interactions, thus forming a more uniform *Timethread Map Description Language* (*TMDL* in section 7.2.3).
- A grammar is not powerful enough to be a complete language by itself. *Semantic rules* are also needed. Defining these rules for TMDL would improve this language's usefulness.
- Tools are still missing. We believe a *compiler* that would generate LOTOS specifications from TMDL descriptions is essential. Tag management and semantic rules also have to be considered.

Long-Term Research Issues

- The definition of *Timethread Correctness Preserving Transformations* (TCPTs) based on LOTOS CPTs would represent a major step towards a more complete methodology.
- Interpretation methods for other target formal semantic model, e.g. Petri nets, would take advantage of the multi-formalism validation allowed by our approach.
- Since we mostly deal with real-time systems, introduction of data and time concepts might be needed sooner or later. Extensions to the LOTOS language concerning time, ADTs, modularity, and typed gates should be studied in the future.
- More complete definitions of path equivalence, conformance, and extension relations between timethreads would be an asset for design validation purpose.
- How not to lose identity of previous timethreads in factored maps (§6.4.1) is a research topic that deserves closer attention.
- The integration of architecture notations and timethread maps needs to be studied for our approach to be useful later in the life-cycle methodology.
- To automate our process, we need tools and GUIs that would generate TMDL descriptions.
- Other real-life case studies could be an excellent way to test and improve the methods defined in this thesis.

References

[Amy 93] D. Amyot, “From Timethreads to LOTOS: A First Pass”, TR-SCE-93-38, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1993)

[BBO 94] F. Bordeleau, R.J.A. Buhr, and A. Obaid, “LOTOS Interpretation of Architecture-Based Design: A Method and a Case Study”, TR-SCE-94-04, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1994)

[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] L. Bernardinello and F. De Cindio, “A Survey of Basic Net Models and Modular Net Classes”, *Lecture Notes in Computer Science #609: Advances in Petri Nets*, Springer-Verlag (1992), 304-351.

[BoA 93] F. Bordeleau and D. Amyot, “LOTOS Interpretation of Timethreads: A Method and a Case Study”, TR-SCE-93-34, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1993)

[BoB 87] T. Bolognesi and E. Brinksma, “Introduction to the ISO Specification Language LOTOS”, *Protocol Specification, Testing and Validation VIII*, North-Holland (1988), 23-73

[BoC 93] T. Bolognesi and G. Ciacco, “Cumulating constraints on the ‘When’ and the ‘What’”, *Proceedings of FORTE’93, Sixth International Conference on Formal Description Techniques*, North-Holland (1993), 435-450

[BoL 94] F. Bordeleau and M. Locas, “Timethread-Centered Design Process: A Study on Transformation Techniques and a Telephone System Case Study”, TR-SCE-94-18, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1994)

[Bor 93] F. Bordeleau, *Visual Descriptions, Formalisms and the Design Process*, MSc. Thesis, School of Computer Science, TR-SCE-93-35, Carleton University, Ottawa, Canada (1993)

[Bou 91] R. Boumezbeur, *Design, Specification, and Validation of Telephony Systems in LOTOS*, MSc. Thesis, TR-91-30, Dept. of Computer Science, University of Ottawa, Ottawa, Canada (1991)

[Bri 88] E. Brinksma, "A Theory for the Derivation of Tests", *Protocol Specification, Testing and Validation VIII*, North-Holland (1988), 63-74.

[BSS 86] E. Brinksma, G. Scollo, and C. Steenbergen, "LOTOS Specifications, their Implementations and their Tests", *Protocol Specification, Testing and Validation VI*, North-Holland (1986), 349-360.

[BuC 92] R.J.A. Buhr and R.S. Casselman, "Architectures with Pictures", *Proceedings of OOPSLA'92*, ACM/SIGPLAN, Vancouver, Canada, (1992), 466-483

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

[BuC 94a] R.J.A. Buhr and R.S. Casselman, "Architecture of the Whole: with Roles and Timethreads", TR-SCE-94-07 (submitted to *IEEE Transactions on Software Engineering*), Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1994)

[BuC 94b] R.J.A. Buhr and R.S. Casselman, "Timethread-Role Maps for Object-Oriented Design of Real-Time-and-Distributed Systems", TR-SCE-94-08 (submitted to *OOPSLA'94*), Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1994)

[Buh 93] R.J.A. Buhr, "Pictures that Play: Design Notations for Real-Time & Distributed Systems", *Software Practice and Experience*, Vol. 23, No. 8, (1993), 895-931

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

[Coo 91] J.E. Cooling, *Software Design for Real-Time Systems*, Chapman and Hall, London, U.K. (1991)

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

[EhM 85] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1 (Equations and Initial Semantics)*, Springer-Verlag (1985)

[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 & Computer Engineering,Carleton University,Ottawa,Canada (1993)

[Ghr 92] B. Ghribi, *A Model Checker for LOTOS*, MSc. Thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada (1992)

[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)

[Hoa 85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, U.K. (1985)

[ISO 88] ISO, Information Processing Systems, Open Systems Interconnection, "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", DIS 8807 (1988)

[JaA 92] I. Jacobson et. al., *Object-Oriented Software Engineering (A Use Case Driven Approach)*, ACM Press, Addison-Wesley (1992)

[Jac 93] R. Jacobs, “Architectural Framework for the Telepresence Voice Server”, Ontario Telepresence Project, Technical Report OTP-93-03 (1993)

[KhB 92] F. Khendek and 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 Time-Threads”, Real-Time and Distributed Systems Group, Carleton University, Ottawa, Canada (1992)

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

[Led 91] G. Leduc, “Conformance Relation, Associated Equivalence, and New Canonical Tester in LOTOS”, *Protocol Specification, Testing and Validation XI*, North-Holland (1991), 249-264

[LFH 91] L. Logrippo, M. Faci, and M. Haj-Hussein, “An Introduction to LOTOS: Learning by Examples”, *Computer Network and ISDN Systems*, No. 23, North-Holland (1992), 325-342

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

[Man 93] M. Mantei, “Telepresence User Interface Design Issues and Solutions”, Ontario Telepresence Project, Technical Report OTP-93-02 (1993)

[Mil 80] R. Milner, “A Calculus of Communicating Systems”, *Lecture Notes in Computer Science*, Vol. 92, Springler-Verlag (1980)

[Mil 92] T. Milligan, “The telepresence Integrated Interactive Intermedia Facility (iiif)”, Ontario Telepresence Project, Technical Report OTP-93-04 (1992)

[MuA 89] J.D. Musa and A.F. Ackerman, “Quantifying Software Validation: When to Stop Testing”, *IEEE Software*, 0740-7459/89/0500/0019, USA (May 1989), 19-27

[Mye 79] G. J. Myers, “The Art of Software Testing”, John Wiley & Sons Inc., USA (1979)

[NuP 93] K. Nursimulu and R.L. Probert, “Cause-Effect Validation of Telecommunications Service Requirements”, TR-93-15, Dept. of Computer Science, University of Ottawa, Ottawa, Canada (1993)

[Pfl 92] S.L. Pfleeger, “Measuring Software Reliability”, *IEEE Spectrum*, 0018-9325/92, USA (August 1992), 56-60

[Pre 87] R.S. Pressman, “Software Engineering: A Practitioner’s Approach”, McGraw-Hill, USA (1987)

[Pro 92] R.L. Probert, “Software Testing: Theory and Practice”, CSI 2511 course notes, Dept. of Computer Science, University of Ottawa, Ottawa, Canada (1992)

[QuA 92] J. Quemada and A. Azcorra, “Structuring Protocols using Exceptions in a LOTOS Extension”, Technical report, DIT-UPM, Madrid, Spain (1992)

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

[Sch 93] J. Schot, “Introduction to LOTOS (ISO 8807) and its Use”, Tutorial Notes of *FORTE’93, Sixth International Conference on Formal Description Techniques*, North-Holland (1993), 141-196

[Tur 90] K.J. Turner, *Proceedings of FORTE'90, Second International Conference on Formal Description Techniques*, North-Holland (1993), 117-133

[Tur 92] K.J. Turner, *Using Formal Description Techniques; An Introduction to ESTELLE, LOTOS and SDL*, Wiley Publishers, U.K. (1992)

[Tur 93] K.J. Turner, "An Engineering Approach to Formal Methods", *Protocol Specification, Testing and Validation XIII*, North-Holland (1993), 165-184.

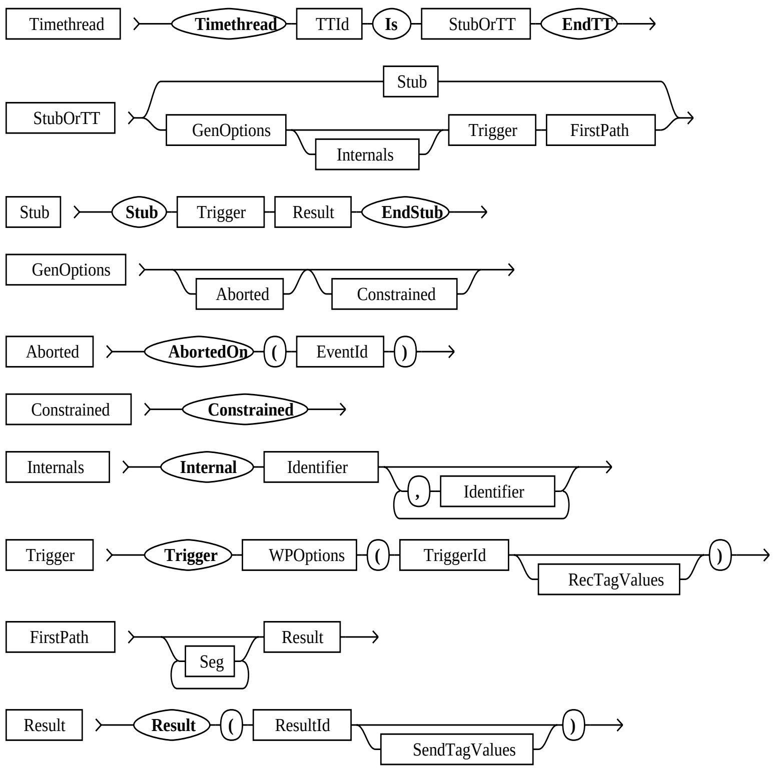
[ViB 91] M. Vigder and R.J.A. Buhr, "Using LOTOS in a Design Environment", *Proceedings of FORTE'91, Fourth International Conference on Formal Description Techniques*, North-Holland (1991), 1-14

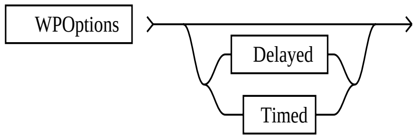
[Vig 92] M. Vigder, *Integrating Formal Techniques into the Design of Concurrent Systems*, Ph.D. Thesis OCIEE-92-03, Dept. Systems and Computer Engineering, Carleton University, Ottawa, Canada (1992)

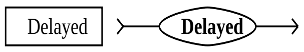
[VSS 91] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma, "Specification Styles in Distributed Systems Design and Verification", *Theoretical Comp. Sci. '89* (1991), 179-206

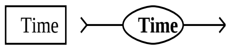
Appendices

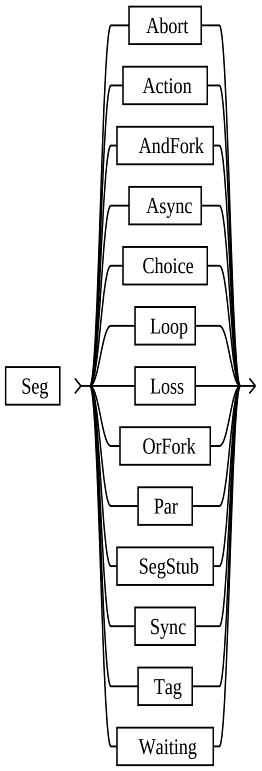
Appendix A. STDL Syntax Diagrams

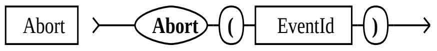


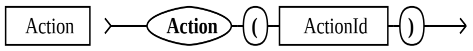

WPOptions
Delayed
Timed

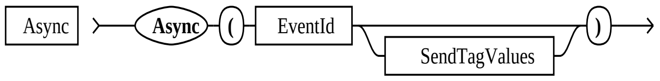

Delayed
Delayed

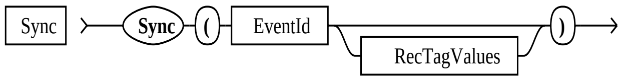

Time
Time

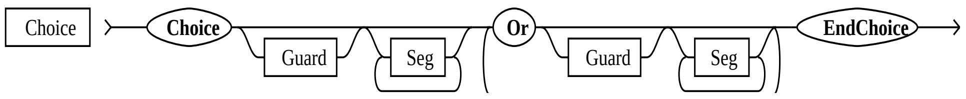

Seg
Abort
Action
AndFork
Async
Choice
Loop
Loss
OrFork
Par
SegStub
Sync
Tag
Waiting

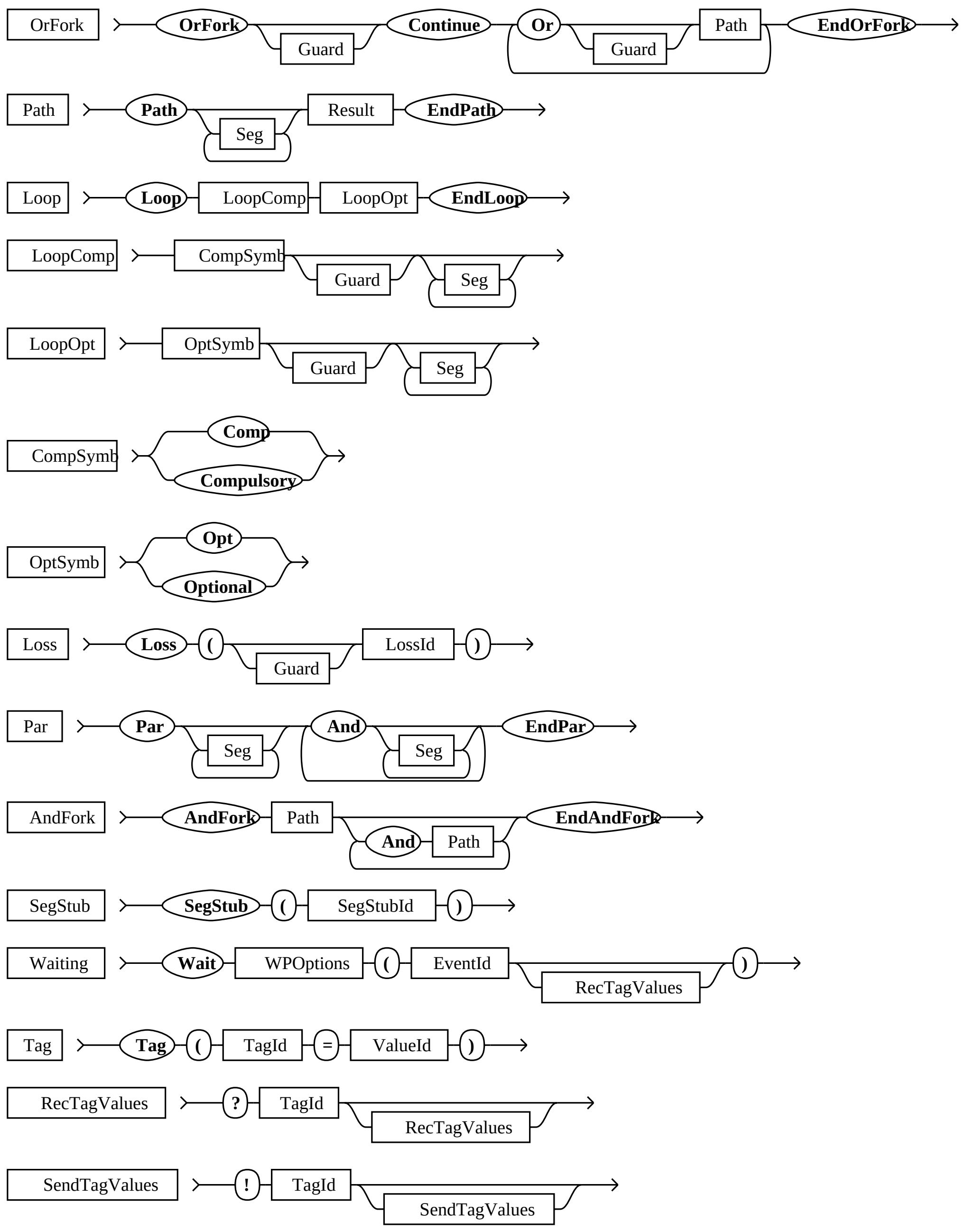

Abort
Abort
(
EventId
)

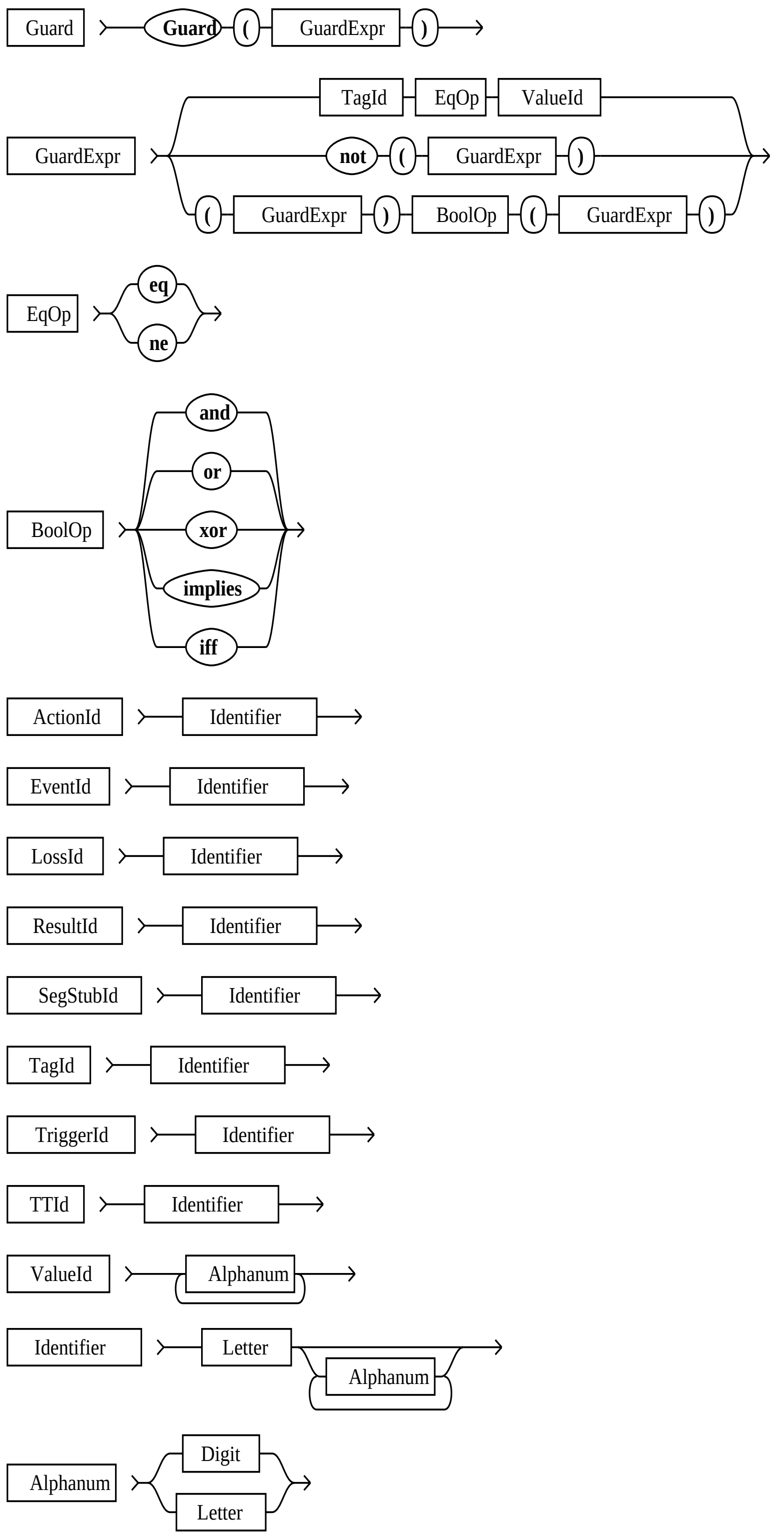

Action
Action
(
ActionId
)


Async
Async
(
EventId
SendTagValues
)


Sync
Sync
(
EventId
RecTagValues
)


Choice
Choice
Guard
Seg
Or
Guard
Seg
EndChoice


OrFork
OrFork
Guard
Continue
Or
Guard
Path
EndOrFork
Path
Path
Seg
Result
EndPath
Loop
Loop
LoopComp
LoopOpt
EndLoop
LoopComp
CompSymb
Guard
Seg
LoopOpt
OptSymb
Guard
Seg
CompSymb
Comp
Compulsory
OptSymb
Opt
Optional
Loss
Loss
(
Guard
LossId
)
Par
Par
Seg
And
Seg
EndPar
AndFork
AndFork
Path
And
Path
EndAndFork
SegStub
SegStub
(
SegStubId
)
Waiting
Wait
WPOptions
(
EventId
RecTagValues
)
Tag
Tag
(
TagId
=
ValueId
)
RecTagValues
?
TagId
RecTagValues
SendTagValues
!
TagId
SendTagValues


Guard
Guard
(
GuardExpr
)
GuardExpr
TagId
EqOp
ValueId
not
(
GuardExpr
)
(
GuardExpr
)
BoolOp
(
GuardExpr
)
EqOp
eq
ne
BoolOp
and
or
xor
implies
iff
ActionId
Identifier
EventId
Identifier
LossId
Identifier
ResultId
Identifier
SegStubId
Identifier
TagId
Identifier
TriggerId
Identifier
TTId
Identifier
ValueId
Alphanum
Identifier
Letter
Alphanum
Alphanum
Digit
Letter

Appendix B. Specification Traveler_Level3

The following specification is the level 3 interpretation of the *Traveler System*. The modifications done to the level 1 specification of chapter 5 to transform it to a level 3 specification are *italicized*. The transformation of processes *Traveler* ans *Plane* are straightforward, but processes *Dispatcher* and *Cab* need to have a more complex mechanism to manage constrained-start timethread in a recursive environment.

```
(* Traveler_Level3; Daniel Amyot, March 29, 1994 *)
(* Level 3 specification of the Traveler system *)

specification Traveler_Level3[Tnew  (* New traveler wants to travel *),
                              Tdest (* Traveler arrives to destination *) ] : noexit

behaviour (* Structure obtained from the LARG *)

hide  (* hidden interactions *)
    TphoneD,        (* Traveler phones Dispatcher for a cab *)
    TgetinC,        (* Traveler gets in the cab *)
    TCride,         (* Traveler and cab ride *)
    TgetoutC,       (* Traveler gets out the cab *)
    TgetonP,        (* Traveler gets on the plane *)
    TPflight,       (* Traveler and plane flight *)
    TgetoffP,       (* Traveler gets off the plane *)
    Din,            (* Dispatcher is in the office *)
    DaskC,          (* Dispatcher asks for a cab *)
    Dout,           (* Dispatcher is not in the office *)
    Cin,            (* Taxi driver in the cab *)
    Cout,           (* Taxi driver not in the cab *)
    Pready,         (* Plane is ready *)
    Phangar         (* Plane goes to the hangar *)

in

    Traveler[Tnew, TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPflight, TgetoffP, Tdest]
    |[TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPflight, TgetoffP]|
    (
        Plane[Pready, TgetonP, TPflight, TgetoffP, Phangar]
        |||
        (
            Dispatcher[Din, TphoneD, DaskC, Dout]
            |[DaskC]|
            Cab[Cin, DaskC, TgetinC, TCride, TgetoutC, Cout]
        )
    )

where

(*  Local hidden actions:                      *)
(*  --------------------                       *)
(*  Traveler: Tairport                         *)
(*  Plane:                                     *)
(*  Dispatcher: DlookforC, Dfillstats, Dready  *)
(*  Cab: CgoD, Cgarage                         *)


(* Timethread Traveler *)
    process Traveler[Tnew, TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPflight, TgetoffP
                     Tdest] : noexit :=
        hide Tairport in  (* hidden action *)
        Tnew;
        (
            TphoneD; stop  (* in-passing interaction *)
            |||
            (
                TgetinC;   (* rest of the path *)
                TCride;
                TgetoutC;
                Tairport;
                TgetonP;
                TPflight;
                TgetoffP;
                Tdest; stop
            )
            |||
            (* recursion for level 3 *)
            Traveler[Tnew, TphoneD, TgetinC, TCride, TgetoutC, TgetonP, TPflight, TgetoffP,
                    Tdest]
        )
    endproc (* Traveler *)

(*-------------------------------------------------------*)

(* Timethread Dispatcher *)
    process Dispatcher[Din, TphoneD, DaskC, Dout] : noexit :=
        (* hidden actions *)
        hide
            DlookforC,   (* Dispatcher looks for a cab *)
            Dfillstats,  (* Dispatcher fills statistics *)
            Dready,      (* Dispatcher is ready for next traveler *)
            SyncCS       (* Internal synchronization for level 3 constrained start *)
        in
        DispatcherWP_CS[Din, SyncCS]
        |[SyncCS]|
        DispatcherSub[SyncCS, TphoneD, DlookforC, DaskC, Dfillstats, Dready, Dout]
        where

        (* Waiting Place Machinery *)
        process DispatcherWP_CS[Din, SyncCS] : noexit :=
            Din;
            (
                SyncCS; stop  (* Allows one token to go *)
                |||
                DispatcherWP_CS[Din, SyncCS]  (* Accumulation of Din *)
            )
        endproc (* DispatcherWP_CS *)

        (* Rest of the timethread *)
        process DispatcherSub[SyncCS, TphoneD, DlookforC, DaskC, Dfillstats, Dready, Dout]
                : noexit :=
            SyncCS; DispatcherLoop[SyncCS, TphoneD, DlookforC, DaskC,Dfillstats,Dready,Dout]
            where

            process DispatcherLoop[SyncCS, TphoneD, DlookforC, DaskC,Dfillstats,Dready,Dout]
                    : noexit :=
                (* Compulsory segment *)
                TphoneD;
                DlookforC;
                (
                    DaskC; stop  (* in-passing interaction *)
                    |||
                    Dfillstats;
                    (
                        (* Optional segment *)
                        Dready;
                        DispatcherLoop[SyncCS,TphoneD,DlookforC,DaskC,Dfillstats,Dready,Dout]
                        []
                        (* Exit Loop *)
                        Dout;
                        DispatcherSub[SyncCS,TphoneD,DlookforC,DaskC,Dfillstats,Dready,Dout]
                    )
                )
            endproc (* DispatcherLoop *)
        endproc (* DispatcherSub *)
    endproc (* Dispatcher *)

(*-------------------------------------------------------*)

(* Timethread Cab *)
    process Cab[Cin, DaskC, TgetinC, TCride, TgetoutC, Cout] : noexit :=
        (* hidden actions *)
        hide
            CgoD,    (* Cab goes to wait the dispatcher *)
            Cgarage, (* Cab goes to the garage *)
            SyncCS   (* Internal synchronization for level 3 constrained start *)
        in
        CabWP_CS[Cin, SyncCS]
        |[SyncCS]|
        Cabsub[SyncCS, DaskC, TgetinC, TCride, TgetoutC, CgoD, Cgarage, Cout]
        where

        (* Waiting Place Machinery *)
        process CabWP_CS[Cin, SyncCS] : noexit :=
            Cin;
            (
                SyncCS; stop  (* Allows one token to go *)
                |||
                CabWP_CS[Cin, SyncCS]  (* Accumulation of Cin *)
            )
        endproc (* DispatcherWP_CS *)

        (* Rest of the timethread *)
        process Cabsub[SyncCS, DaskC, TgetinC, TCride, TgetoutC, CgoD, Cgarage, Cout]
                : noexit :=
            SyncCS; CabLoop[SyncCS, DaskC, TgetinC, TCride, TgetoutC, CgoD, Cgarage, Cout]
            where

            process CabLoop[SyncCS, DaskC, TgetinC, TCride, TgetoutC, CgoD, Cgarage, Cout]
                    : noexit :=
                (* Compulsory segment *)
                DaskC;
                TgetinC;
                TCride;
                TgetoutC;
                (
                    (* Optional segment *)
                    CgoD; CabLoop[SyncCS, DaskC, TgetinC, TCride, TgetoutC,CgoD,Cgarage,Cout]
                    []
                    (* Exit Loop *)
                    Cgarage;
                    Cout; Cabsub[SyncCS, DaskC, TgetinC, TCride,TgetoutC,CgoD, Cgarage, Cout]
                )
            endproc (* CabLoop *)
        endproc (* Cabsub *)
    endproc (* Cab *)

(*-------------------------------------------------------*)

(* Timethread_Plane *)
    process Plane[Pready, TgetonP, TPflight, TgetoffP, Phangar] : noexit :=
        (* no hidden action in the timethread *)
        Pready;
        (
            TgetonP;
            TPflight;
            TgetoffP;
            Phangar; stop
            |||
            (* recursion for level 3 *)
            Plane[Pready, TgetonP, TPflight, TgetoffP, Phangar]
        )
    endproc (* Plane *)
endspec (* Specification Traveler_Level3 *)
```

Specification 11: Traveler System, Level 3

Appendix C. Specification Telepresence

This specification is the level 1 interpretation (without recursion) of the *Telepresence System* (chapter 6).

```
(* Specification Telepresence, 14 April 1994 *)
(* Level 1 specification, without recursion *)

specification Telepresence[ Contact,  (* Initiator want to contact responder *)
                            Report,   (* Result of a contact *)
                            Close,    (* Close user's door *)
                            Open,     (* Open user's door *)
                            Terminate,(* Initiator terminates a connection *)
                            Next      (* Get ready for next connection *) ] : noexit

library
   Boolean, NaturalNumber
endlib

(* Tag ADT definition. All possible tags are enumerated here. *)
type Tag is Boolean, NaturalNumber
sorts Tag
opns No, Yes,                    (* Tags *)
     Success, Denied, TimeOut,
     Open, Close,
     Access,
     OK, TOut      : -> Tag
     N         : Tag -> Nat      (* Tag-to-Nat function *)
     _eq_,
     _ne_ : Tag, Tag -> Bool     (* Tag equivalence *)
eqns forall x, y: Tag
    ofsort Nat
    N(No)      = 0;
    N(Yes)     = Succ(N(No));
    N(Success) = Succ(N(Yes));
    N(Denied)  = Succ(N(Success));
    N(TimeOut) = Succ(N(Denied));
    N(Open)    = Succ(N(TimeOut));
    N(Close)   = Succ(N(Open));
    N(Access)  = Succ(N(Close));
    N(OK)      = Succ(N(Access));
    N(TOut)    = Succ(N(OK));
    ofsort Bool
    x eq y     = N(x) eq N(y);
    x ne y     = not(x eq y);
endtype

behaviour (* Architecture obtained from the LARG *)

hide  (* hidden interactions *)
   Knock,        (* Send a checkdoor request *)
   DoorState,    (* Check the responder's door state *)
   DoorChecked,  (* The responder's door has been checked *)
   Transmit,     (* Begin transmission of voice and images *)
   Receive,      (* Start the reception *)
   SendState,    (* Indicate to the responder wether the initiator disconnects *)
   EndSend,      (* The transmission is ended *)
   RecState,     (* Indicate wether the initiator asked to terminate *)
   Played,       (* Voice and image messages were played *)
   EndRec        (* The connection has terminated *)

```

```
in

   (
      (
         Connection[Contact, Knock, DoorChecked, EndRec, Report, Next]
         |||
         Transmission[Transmit, Receive, SendState, EndSend]
      )
      |[Receive, SendState, EndRec]|
      (
         Disconnection[Terminate, RecState]
         |[RecState]|
         Reception[Receive, RecState, Played, SendState, EndRec]
      )
   )
   |[Knock, DoorChecked, Transmit]|
   (
      Knocking[Knock, DoorState, Transmit, DoorChecked]
      |[DoorState]|
      (
         Opening[Open, DoorState]
         |||
         Closing[Close, DoorState]
      )
   )

where

(*-------------------------------------------------------*)

(* Timethread Connection *)
   process Connection[Contact, Knock, DoorChecked, EndRec, Report, Next] : noexit :=
      (* hidden gates for time WP and choice *)
      hide
         TimeOutCon,
         SyncTOCon,
         SyncOrCon
      in
      Contact;
      (
         Knock; stop
         |||
         (* Wait Time Mechanism *)
         (
            TimeOutCon;
            SyncTOCon ! TOut ! TOut; (* The 2nd tag is a dummy used for synchro only *)
            stop
            []
            DoorChecked ? Resp: Tag;
            SyncTOCon ! OK ! Resp;
            stop
         )
         |[SyncTOCon]|
         SyncTOCon ? ResultTO: Tag ? Resp: Tag; (* Tags have to follow *)
         (* Choice Mechanism *)
         (
            [ResultTO eq TOut] (* [G1] *) ->
               (let Rpt: Tag = TimeOut in SyncOrCon ! ResultTO ! Resp ! Rpt; stop)
            []
            [(ResultTO eq OK) and (Resp eq Denied)] (* [G2] *) ->
               (let Rpt: Tag = Denied in SyncOrCon ! ResultTO ! Resp ! Rpt; stop)
            []
            [(ResultTO eq OK) and (Resp eq Access)] (* [G3] *) ->
               (let Rpt: Tag = Success in
                  EndRec;
```

```
                SyncOrCon ! ResultTO ! Resp ! Rpt; stop
             )
       )
       |[SyncOrCon]|
       SyncOrCon ? ResultTO: Tag ? Resp: Tag ? Rpt: Tag;
       (
          Report ! Rpt; stop
          |||
          Next; stop
       )
    )
 endproc (* Connection *)

(*-------------------------------------------------------*)

(* Timethread Transmission *)
 process Transmission[Transmit, Receive, SendState, EndSend] : noexit :=
    hide
    (* hidden actions *)
       Signal,    (* Someone is observing *)
       RecVoice,  (* The micro records the voice *)
       RecImage   (* The camera records the image *)
    in
    Transmit; TLoop [Receive, SendState, EndSend, Signal, RecVoice, RecImage]

 where
    process TLoop [Receive, SendState, EndSend, Signal, RecVoice, RecImage] : noexit :=
       (* hidden gate for Par and Signal WP *)
       hide
          SyncAndTra,
          SyncSigTra
       in
       Signal;
       (
          RecImage; SyncAndTra; stop
          |[SyncAndTra]|
          RecVoice; SyncAndTra; stop
       )
       |[SyncAndTra]|
       SyncAndTra;
       (
          Receive; stop
          |||
          (* Waiting place signal management *)
          (
             SendState; SyncSigTra ! Yes; stop
             []
             SyncSigTra ! No; stop
          )
          |[SyncSigTra]|
          SyncSigTra ? Sig: Tag;
          (
             [Sig eq No] (* [G6] *) ->
                (TLoop [Receive, SendState, EndSend, Signal, RecVoice, RecImage])
             []
             [Sig eq Yes] (* [G7] *) -> (EndSend; stop)
          )
       )
    endproc (* TLoop *)
 endproc (* Transmission *)

```

```
(*-------------------------------------------------------*)

(* Timethread Disconnection *)
   process Disconnection[Terminate, RecState] : noexit :=
      Terminate;
      (
         RecState; stop
      )
   endproc (* Disconnection *)

(*-------------------------------------------------------*)

(* Timethread Reception *)
   process Reception[Receive, RecState, Played, SendState, EndRec] : noexit :=
      hide
      (* hidden actions *)
         PlayImage,    (* The monitor shows the image *)
         PlayVoice,    (* The speaker plays the voice *)
         Disconnect,   (* Update disconnection status *)
      (* hidden gates for Par and Signal WP *)
         SyncAndRec,
         SyncSigRec
      in
      Receive;
      (
         (
            PlayImage; SyncAndRec; stop
            |[SyncAndRec]|
            PlayVoice; SyncAndRec; stop
         )
         |[SyncAndRec]|
         SyncAndRec;
         (* Waiting place signal management *)
         (
            RecState; SyncSigRec ! Yes; stop
            []
            SyncSigRec ! No; stop
         )
         |[SyncSigRec]|
         SyncSigRec ? Sig: Tag;
         (
            [Sig eq Yes] (* [G9] *) ->
               (SendState; stop
               |||
               Disconnect;
               EndRec; stop
               )
            []
            [Sig eq No] (* [G8] *) -> (Played; stop)
         )
      )
   endproc (* Reception *)

(*-------------------------------------------------------*)

(* Timethread Knocking *)
   process Knocking[Knock, DoorState, Transmit, DoorChecked] : noexit :=
      (* hidden gates for signal WP *)
      hide
         DoorStateMem  (* Memory cell for internal use *)
      in
      (* Waiting place with memory management *)
      DoorStateMemory [DoorState, DoorStateMem] (Close)
      |[DoorStateMem]|
      Knocking2[Knock, DoorStateMem, Transmit, DoorChecked]
      where

      process Knocking2[Knock, DoorStateMem, Transmit, DoorChecked] : noexit :=
         (* hidden gates for lost and choice *)
         hide
            KnockLost,    (* The knock request is lost *)
            SyncOrKno     (* Internal synchro. for the OR fork *)
         in
         Knock;
         (
            KnockLost; stop
            []
            DoorStateMem ? DS: Tag;
            (
              [DS eq Close] (* [G4] *) ->(let Rep: Tag = Denied in SyncOrKno ! DS ! Rep; stop)
               []
               [DS eq Open] (* [G5] *) ->
                  (let Rep: Tag = Access in
                     Transmit; stop
                     |||
                     SyncOrKno ! DS ! Rep; stop
                  )
            )
            |[SyncOrKno]|
            SyncOrKno ? DS: Tag ? Rep : Tag;
            DoorChecked ! Rep;
            stop
         )
      endproc (* Knocking2 *)

      process DoorStateMemory [DoorState, DoorStateMem] (Mem: Tag): noexit :=
         DoorState ? NewMem: Tag; DoorStateMemory [DoorState, DoorStateMem] (NewMem)
         []
         DoorStateMem ! Mem; DoorStateMemory [DoorState, DoorStateMem] (Mem)
      endproc (* DoorStateMemory *)

   endproc (* Knocking *)

(*----------------------------------------------------------*)

(* Timethread Opening *)
   process Opening[Open, DoorState] : noexit :=
      Open;
      (
         let D: Tag = Open in
         DoorState ! D; stop
      )
   endproc (* Opening *)

(*----------------------------------------------------------*)

(* Timethread Closing *)
   process Closing[Close, DoorState] : noexit :=
      Close;
      (
         let D: Tag = Close in
         DoorState ! D; stop
      )
   endproc (* Closing *)

endspec (* Telepresence *)
```

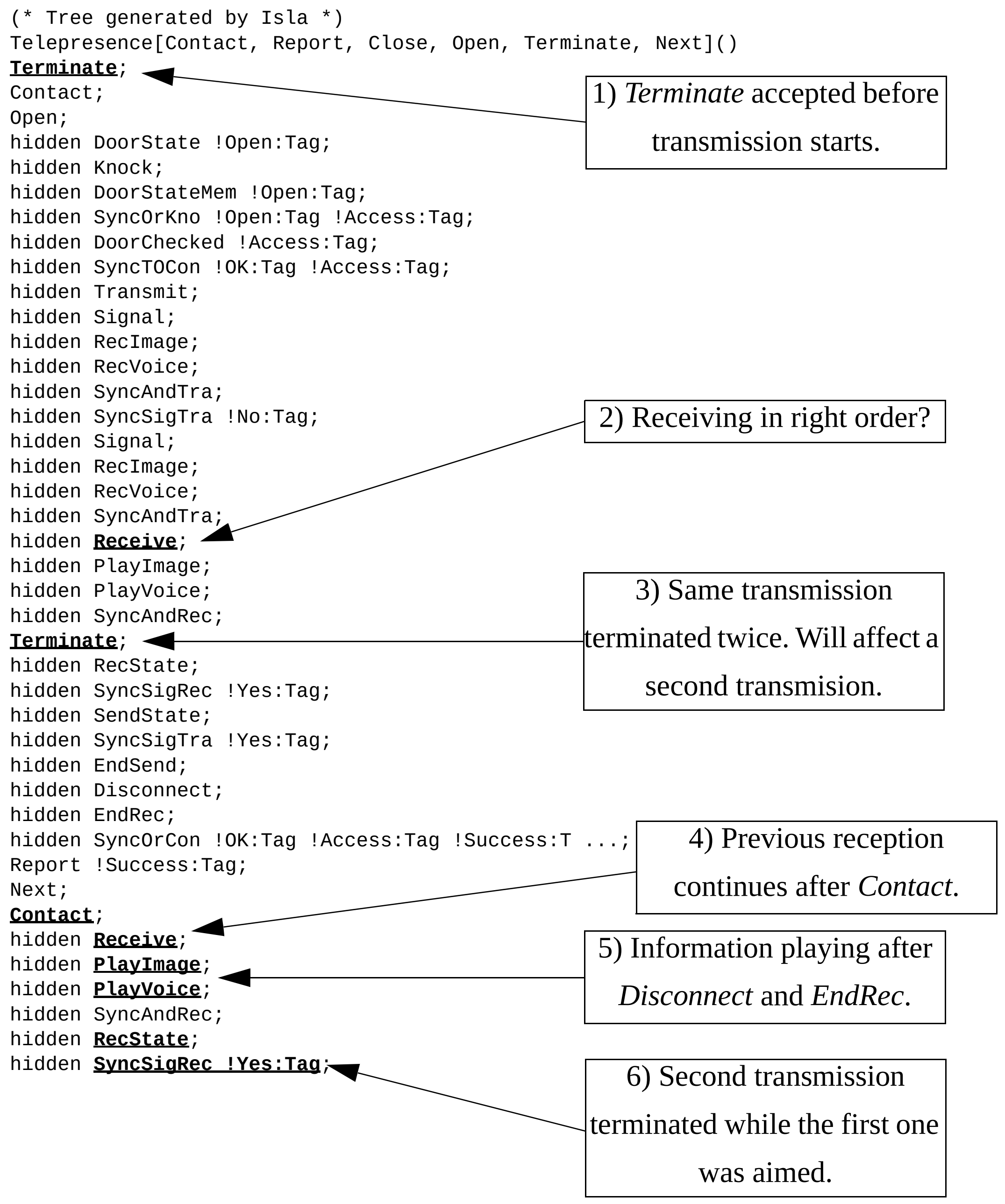
Specification 12: Telepresence System, Level 1 Without Recursion

Appendix D. Validation of Telepresence, Level 1 with Recursion

We present here the validation of the level 1 specification Telepresence (with recursion) against different scenarios. We use the simulation and testing techniques presented in section 5.4.

D.I Simulation

The following sequence of activities is a complex scenario (connection-transmission-connection), obtained with the help of ELUDO, that reveals a few problems associated to the current design.



```
(* Tree generated by Isla *)
Telepresence[Contact, Report, Close, Open, Terminate, Next]()
Terminate;
Contact;
Open;
hidden DoorState !Open:Tag;
hidden Knock;
hidden DoorStateMem !Open:Tag;
hidden SyncOrKno !Open:Tag !Access:Tag;
hidden DoorChecked !Access:Tag;
hidden SyncTOCon !OK:Tag !Access:Tag;
hidden Transmit;
hidden Signal;
hidden RecImage;
hidden RecVoice;
hidden SyncAndTra;
hidden SyncSigTra !No:Tag;
hidden Signal;
hidden RecImage;
hidden RecVoice;
hidden SyncAndTra;
hidden Receive;
hidden PlayImage;
hidden PlayVoice;
hidden SyncAndRec;
Terminate;
hidden RecState;
hidden SyncSigRec !Yes:Tag;
hidden SendState;
hidden SyncSigTra !Yes:Tag;
hidden EndSend;
hidden Disconnect;
hidden EndRec;
hidden SyncOrCon !OK:Tag !Access:Tag !Success:T ...;
Report !Success:Tag;
Next;
Contact;
hidden Receive;
hidden PlayImage;
hidden PlayVoice;
hidden SyncAndRec;
hidden RecState;
hidden SyncSigRec !Yes:Tag;
```

1) *Terminate* accepted before transmission starts.

2) Receiving in right order?

3) Same transmission terminated twice. Will affect a second transmision.

4) Previous reception continues after *Contact*.

5) Information playing after *Disconnect* and *EndRec*.

6) Second transmission terminated while the first one was aimed.

D.II Testing

Two acceptance and one rejection test cases, inspired from the *Connection* use case, are applied to the *Telepresence* specification. We use the testing facilities of LOLA to get meaningful results.

First Acceptance Test Case

This process tests whether the possible *Report*s are *Denied* or *TimeOut* when the responder's door is closed (by default in the specification).

```
process AcceptTest1[Contact, Report, Success]: noexit :=
(* Test for denied or timeout report when the door is closed (default) *)
   Contact;
   (
      Report ! Denied; Success; stop
   []
      Report ! TimeOut; Success; stop
   )
endproc (* AcceptTest1 *)
```

Specification 13: Acceptance test case 1 for level 1 Telepresence System

We use LOLA's command *TestExpand* to test all possible scenarios. The result, expressed below, shows our test case to be a **must pass** test. This indicates a valid behaviour from our specification.

```
lola> TestExpand -1 Success AcceptTest1 -i

 Composing behaviour and test :

    Analysed states       = 139
    Generated transitions = 184
    Duplicated states     = 0
    Deadlocks             = 0

    Process Test = accepttest1
    Test result  = MUST PASS.

                   successes = 46
                       stops = 0
                       exits = 0
               cuts by depth = 0
```

First Rejection Test Case

This second process tests that *Report* cannot output *Success* when the responder's door is closed (it is initially closed in the specification).

```
process RejectTest1[Contact, Report, Fail]: noexit :=
(* Test for denied or timeout report when the door is closed (default) *)
   Contact;
   Report ! Success; Fail; stop
endproc (* RejectTest1 *)
```

Specification 14: Rejection test case 1 for level 1 Telepresence System

The *TestExpand* procedure returns **reject**, meaning that our specification behaves properly.

```
lola> TestExpand -1 Fail RejectTest1 -i

 Composing behaviour and test :

    Analysed states       = 93
    Generated transitions = 92
    Duplicated states     = 0
    Deadlocks             = 31

    Process Test = rejecttest1
    Test result  = REJECT.

                   successes = 0
                       stops = 31
                       exits = 0
               cuts by depth = 0
```

Second Acceptance Test Case

This more complex process tests that, when the responder *Open*s his/her door and the initiator *Contact*s the responder, the *Report* should output a *Success* or possibly a *TimeOut* when the initiator *Terminate*s the connection.

```
process AcceptTest2[Contact, Open, Terminate, Report, Success]: noexit :=
(* Test for success (or timeout) report when the responder opens the door *)
   Open;
   Contact;
   (
      Terminate;
      (
         Report ! Success; Success; stop
         []
         Report ! TimeOut; Success; stop
      )
   []
      Report ! TimeOut; Success; stop
   )
endproc (* AcceptTest2 *)
```

Specification 15: Acceptance test case 2 for level 1 Telepresence System

Since the transition system generated from the *Telepresence* specification is infinite when there is a transmission, we must limit the depth of the search (here to 15 events) of the *TestExpand* procedure. This generates a huge number of executions (509507) that should give us confidence in the result.

```
lola> TestExpand 15 Success AcceptTest2 -i -y

 Composing behaviour and test :

    Analysed states       = 331916
    Generated transitions = 841382
    Duplicated states     = 0
    Deadlocks             = 40

    Process Test = accepttest2
    Test result  = MAY PASS.

    509507 executions analysed:

                   successes = 63617
                       stops = 40
                       exits = 0
               cuts by depth = 445850
```

LOLA outputs **may pass** as a result. On 509507 executions, 63617 were successful, 40 were unsuccessful, and the remaining 445850 were unfinished (they were probably still in the transmission loop).

We can look for specific instances of successful and unsuccessful executions with the *OneExpand* command of LOLA. By using different seeds, we were able to find an instance of a *Success* report and one of a *TimeOut* report (fig. 93). We also found one unsuccessful execution where the deadlock was caused by a closed door. Although the responder *Open*ed the door before the *Contact* of the initiator, the door did not change its internal state before it was checked by the knocking request. This race condition is therefore due to time lapsed between *Open* and *DoorState*.

LOLA also allows users to take a closer look to its diagnostics. The option `-s` of *TestExpand* output all traces leading to a deadlock, in the form of a monolithic LOTOS tree with choices and action prefixes only. Other options output the other types of traces:

- `-d` : traces cut by depth search.
- `-a` : traces leading to the success event.
- `-e` : traces leading to ***exit***.

```
(* Reached a Success report *)
OneExpand -1 Success AcceptTest2 182 -v -i

 Composing behaviour and test :

    1  open;
    2  contact;
    3  i; (* doorstate ! open *)
    4  i; (* knock *)
    5  i; (* doorstatemem ! open *)
    6  i; (* syncorkno ! open ! access *)
    7  i; (* transmit *)
    8  i; (* doorchecked ! access *)
    9  i; (* synctocon ! ok ! access *)
   10  terminate;
   11  i; (* signal *)
   12  i; (* recvoice *)
   13  i; (* recimage *)
   14  i; (* syncandtra *)
   15  i; (* syncsigtra ! no *)
   16  i; (* signal *)
   17  i; (* recvoice *)
   18  i; (* receive *)
   19  i; (* playimage *)
   20  i; (* playvoice *)
   21  i; (* syncandrec *)
   22  i; (* recimage *)
   23  i; (* recstate *)
   24  i; (* syncandtra *)
   25  i; (* syncsigrec ! yes *)
   26  i; (* syncsigtra ! no *)
   27  i; (* signal *)
   28  i; (* disconnect *)
   29  i; (* recvoice *)
   30  i; (* recimage *)
   31  i; (* endrec *)
   32  i; (* syncorcon! ok !access !success *)
   33  i; (* receive *)
   34  report  ! success;
   35  i; (* syncandtra *)
   36  i; (* syncsigtra ! no *)
   37  success;

    Process Test = accepttest2
    Test result  = SUCCESSFUL EXECUTION.

       Transitions generated = 37
```

```
(* Reached a TimeOut report *)
OneExpand -1 Success AcceptTest2 144 -v -i

 Composing behaviour and test :

    1  open;
    2  contact;
    3  terminate;
    4  i; (* knock *)
    5  i; (* doorstate ! open *)
    6  i; (* knocklost *)
    7  i; (* timeoutcon *)
    8  i; (* synctocon ! tout ! tout *)
    9  i; (* syncorcon !tout !tout !timeout *)
   10  report  ! timeout;
   11  success;

    Process Test = accepttest2
    Test result  = SUCCESSFUL EXECUTION.

       Transitions generated = 11
```

```
(* Deadlock caused by a closed door *)
OneExpand -1 Success AcceptTest2 16 -v -i

 Composing behaviour and test :

    1  open;
    2  contact;
    3  terminate;
    4  i; (* knock *)
    5  i; (* doorstatemem ! close *)
    6  i; (* syncorkno ! close ! denied *)
    7  i; (* doorchecked ! denied *)
    8  i; (* doorstate ! open *)
    9  i; (* synctocon ! ok ! denied *)
   10  i; (* syncorcon ! ok !denied !denied *)
   11  stop

    Process Test = accepttest2
    Test result  = REJECTED EXECUTION.

       Transitions generated = 11
```

Figure 93: Scenarios generated with OneExpand for the Telepresence specification

Appendix E. Specification Telepresence_1SUD

This specification is the level 1 interpretation (with recursion) of the *Telepresence System* where functionalities of both roles (Initiator/Responder) are within one SUD (chapter 6).

```
(* Specification Telepresence_1SUD, 28 May 1994 *)
(* Level 1 specification, with recursion *)

specification Telepresence_1SUD [ Contact,  (* Initiator want to contact responder *)
                                  Report,   (* Result of a contact *)
                                  Close,    (* Close user's door *)
                                  Open,     (* Open user's door *)
                                  Terminate,(* Initiator terminates a connection *)
                                  Next,     (* Get ready for next connection *)
                                  DIn, DOut (* Incoming/Outgoing data*) ] : noexit

library
   Boolean, NaturalNumber
endlib

(* Tag ADT definition. All possible tags are enumerated here. *)
type Tag is Boolean, NaturalNumber
sorts Tag
opns No, Yes,                    (* Tags *)
     Success, Denied, TimeOut,
     Open, Close,
     Access,
     OK, TOut,
     RDataIn, RDataOut,
     KDataIn, KDataOut,
     dummy   : -> Tag
     N        : Tag -> Nat      (* Tag-to-Nat function *)
     _eq_,
     _ne_ : Tag, Tag -> Bool     (* Tag equivalence *)
eqns forall x, y: Tag
    ofsort Nat
    N(No)       = 0;             (* Tag-to-Natural Mapping *)
    N(Yes)      = Succ(N(No));
    N(Success)  = Succ(N(Yes));
    N(Denied)   = Succ(N(Success));
    N(TimeOut)  = Succ(N(Denied));
    N(Open)     = Succ(N(TimeOut));
    N(Close)    = Succ(N(Open));
    N(Access)   = Succ(N(Close));
    N(OK)       = Succ(N(Access));
    N(TOut)     = Succ(N(OK));
    N(RDataIn)  = Succ(N(TOut));
    N(RDataOut) = Succ(N(RDataIn));
    N(KDataIn)  = Succ(N(RDataOut));
    N(KDataOut) = Succ(N(KDataIn));
    N(dummy)    = Succ(N(KDataOut));
    ofsort Bool
    x eq y     = N(x) eq N(y);
    x ne y     = not(x eq y);
endtype

behaviour (* Architecture obtained from the LARG *)

hide  (* hidden interactions *)
   Knock,        (* Send a checkdoor request *)
   DoorState,    (* Check the responder's door state *)
   DoorChecked,  (* The responder's door has been checked *)
   Transmit,     (* Begin transmission of voice and images *)
   Receive,      (* Start the reception *)
   SendState,    (* Indicate to the responder wether the initiator disconnects *)
   EndSend,      (* The transmission is ended *)
   RecState,     (* Indicate wether the initiator asked to terminate *)
   Played,       (* Voice and image messages were played *)
   EndRec        (* The connection has terminated *)

in

   (
      (
         Connection[Contact, Knock, DoorChecked, EndRec, Report, Next]
         |[Knock]|
         Knocking[Knock, DOut]
      )
      |[EndRec]|
      (
         Transmission[Transmit, Receive, SendState, EndSend]
         |[Receive]|
         Reception[Receive, DOut]
      )
   )
   |[Transmit, DoorChecked, SendState]|
   (
      (
         Data[DIn, DOut, SendState, DoorChecked, DoorState,Transmit, RecState,Played, EndRec]
         |[RecState]|
         Disconnection[Terminate, RecState]
      )
      |[DoorState]|
      (
         Opening[Open, DoorState]
         |||
         Closing[Close, DoorState]
      )
   )

where

(*--------------------------------------------------------*)

(* Timethread Connection *)
   process Connection[Contact, Knock, DoorChecked, EndRec, Report, Next] : noexit :=
      (* hidden gates for time WP and choice *)
      hide
         TimeOutCon,
         SyncTOCon,
         SyncOrCon
      in
      Contact;
      (
         Knock; stop
         |||
         (* Wait Time Mechanism *)
         (
            TimeOutCon;
            SyncTOCon ! TOut ! dummy; (* The 2nd tag is a dummy used for synchro only *)
            stop
            []
            DoorChecked ? Resp: Tag;
            SyncTOCon ! OK ! Resp;
            stop
         )
         |[SyncTOCon]|
         SyncTOCon ? ResultTO: Tag ? Resp: Tag; (* Tags have to follow *)
         (* Choice Mechanism *)
         (
            [ResultTO eq TOut] ->  (* [G1] *)
               (let Rpt: Tag = TimeOut in SyncOrCon ! ResultTO ! Resp ! Rpt; stop)
            []
            [(ResultTO eq OK) and (Resp eq Denied)] ->  (* [G2] *)
               (let Rpt: Tag = Denied in SyncOrCon ! ResultTO ! Resp ! Rpt; stop)
            []
            [(ResultTO eq OK) and (Resp eq Access)] ->  (* [G3] *)
               (let Rpt: Tag = Success in
                  EndRec;
                  SyncOrCon ! ResultTO ! Resp ! Rpt; stop
               )
         )
         |[SyncOrCon]|
         SyncOrCon ? ResultTO: Tag ? Resp: Tag ? Rpt: Tag;
         (
            Report ! Rpt; stop
            |||
            Next; Connection[Contact, Knock, DoorChecked, EndRec, Report, Next]
         )
      )
   endproc (* Connection *)

(*--------------------------------------------------------*)

(* Timethread Transmission *)
   process Transmission[Transmit, Receive, SendState, EndSend] : noexit :=
      hide
      (* hidden actions *)
         Signal,    (* Someone is observing *)
         RecVoice,  (* The micro records the voice *)
         RecImage   (* The camera records the image *)
      in
      Transmit; TLoop [Transmit, Receive, SendState, EndSend, Signal, RecVoice, RecImage]

   where
      process TLoop [Transmit, Receive,SendState,EndSend, Signal,RecVoice,RecImage]: noexit:=
         (* hidden gate for Par and Signal WP *)
         hide
            SyncAndTra,
            SyncSigTra
         in
         Signal;
         (
            RecImage; SyncAndTra; stop
            |[SyncAndTra]|
            RecVoice; SyncAndTra; stop
         )
         |[SyncAndTra]|
         SyncAndTra;
         (
            Receive; stop
            |||
            (* Waiting place signal management *)
            (
               SendState; SyncSigTra ! Yes; stop
               []
               SyncSigTra ! No; stop
            )
            |[SyncSigTra]|
            SyncSigTra ? Sig: Tag;
            (
               [Sig eq No] ->  (* [G6] *)
                  (TLoop [Transmit, Receive, SendState, EndSend, Signal, RecVoice, RecImage])
               []    (* Next guard is [G7] *)
               [Sig eq Yes] -> (EndSend; Transmission[Transmit, Receive, SendState, EndSend])
            )
         )
      endproc (* TLoop *)
   endproc (* Transmission *)

(*-------------------------------------------------------*)

(* Timethread Disconnection *)
   process Disconnection[Terminate, RecState] : noexit :=
      Terminate;
      (
         RecState; Disconnection[Terminate, RecState]
      )
   endproc (* Disconnection *)

(*-------------------------------------------------------*)

(* Timethread Reception *)
   process Reception[Receive, DOut] : noexit :=
      Receive;
      (
         let oP: Tag = RDataIn in DOut ! oP ! dummy; Reception[Receive, DOut]
      )
   endproc (* Reception *)

(*-------------------------------------------------------*)

(* Timethread Knocking *)
   process Knocking[Knock, DOut] : noexit :=
      hide
         KnockLost    (* The knock request is lost *)
      in
      Knock;
      (
         KnockLost; Knocking[Knock, DOut]
         []
         (let oP: Tag = KDataIn in DOut ! oP ! dummy; Knocking[Knock, DOut])
      )
   endproc (* Knocking *)

(*-------------------------------------------------------*)

(* Timethread Opening *)
   process Opening[Open, DoorState] : noexit :=
      Open;
      (
         let D: Tag = Open in
         DoorState ! D; Opening[Open, DoorState]
      )
   endproc (* Opening *)

(*-------------------------------------------------------*)

```

```
(* Timethread Closing *)
   process Closing[Close, DoorState] : noexit :=
      Close;
      (
         let D: Tag = Close in
         DoorState ! D; Closing[Close, DoorState]
      )
   endproc (* Closing *)

(*-------------------------------------------------------*)

   process Data[DIn, DOut, SendState, DoorChecked, DoorState, Transmit, RecState, Played,
                EndRec]: noexit :=

      (* hidden gates for signal WP *)
      hide
         DoorStateMem  (* Memory cell for internal use *)
      in
      (* Waiting place with memory management *)
      DoorStateMemory [DoorState, DoorStateMem] (Close)
      |[DoorStateMem]|
      Data2[DoorStateMem,DIn, DOut,SendState, DoorChecked, Transmit,RecState, Played,EndRec]
      where

      process Data2[DoorStateMem, DIn, DOut, SendState, DoorChecked, Transmit, RecState,
                    Played, EndRec]: noexit :=
         (* hidden gates for Choice, Par and Signal WP *)
         hide
            SyncOrKno,    (* Internal synchro. for the OR fork *)
            SyncAndRec,
            SyncSigRec,
        (* hidden actions *)
            PlayImage,    (* The monitor shows the image *)
            PlayVoice,    (* The speaker plays the voice *)
            Disconnect    (* Update disconnection status *)
         in

         DIn ? iP: Tag ? iRep :Tag;
         (
            [(iP ne KDataOut) and (iP ne RDataOut)] ->  (* [Gadded] *)
               (
                  [iP eq RDataIn] ->     (* [G10] *)  (* Body of Reception *)
                     (
                        (
                           PlayImage; SyncAndRec; stop
                           |[SyncAndRec]|
                           PlayVoice; SyncAndRec; stop
                        )
                        |[SyncAndRec]|
                        SyncAndRec;
                        (* Waiting place signal management *)
                        (
                           RecState; SyncSigRec ! Yes; stop
                           []
                           SyncSigRec ! No; stop
                        )
                        |[SyncSigRec]|
                        SyncSigRec ? Sig: Tag;
                        (
                           [Sig eq Yes] ->   (* [G9] *)
                              (let oP: Tag = RDataOut in
                                  DOut ! oP ! dummy;
                                  Data2[DoorStateMem,DIn,DOut,SendState,DoorChecked,Transmit,
                                        RecState, Played, EndRec]
                              |||
                              Disconnect;
                              EndRec; stop
                              )
                           []
                           [Sig eq No]  (* [G8] *) ->
                             ( Played; Data2[DoorStateMem, DIn, DOut, SendState, DoorChecked,
                                              Transmit, RecState, Played, EndRec])
                        )
                     )
                  []
                  [iP eq KDataIn] ->  (* [G13] *)   (* Body of Knocking *)
                     (
                        DoorStateMem ? DS: Tag;
                        (
                           [DS eq Close] ->  (* [G4] *)
                              (let oRep: Tag = Denied in SyncOrKno ! DS ! oRep; stop)
                           []
                           [DS eq Open] ->   (* [G5] *)
                              (let oRep: Tag = Access in
                                 Transmit; stop
                                 |||
                                 SyncOrKno ! DS ! oRep; stop
                              )
                        )
                        |[SyncOrKno]|
                        SyncOrKno ? DS: Tag ? oRep : Tag;
                        (let oP: Tag = KDataOut in
                           DOut ! oP ! oRep;
                           Data2[DoorStateMem, DIn, DOut, SendState, DoorChecked, Transmit,
                                 RecState, Played, EndRec])
                     )
               )
            []
            [iP eq KDataOut] ->  (* [G11] *)  (* End of Knocking *)
               (DoorChecked ! iRep;
                Data2[DoorStateMem, DIn, DOut, SendState, DoorChecked, Transmit, RecState,
                      Played, EndRec])
            []
            [iP eq RDataOut] ->  (* [G12] *)  (* End of Reception *)
               (SendState;
                Data2[DoorStateMem, DIn, DOut, SendState, DoorChecked, Transmit, RecState,
                      Played, EndRec])
         )
      endproc (* Data2 *)

      process DoorStateMemory [DoorState, DoorStateMem] (Mem: Tag): noexit :=
         DoorState ? NewMem: Tag; DoorStateMemory [DoorState, DoorStateMem] (NewMem)
         []
         DoorStateMem ! Mem; DoorStateMemory [DoorState, DoorStateMem] (Mem)
      endproc (* DoorStateMemory *)

   endproc (* Data *)

endspec (* Telepresence_1SUD *)
```

Specification 16: Telepresence System (1 SUD), Level 1 With Recursion

Appendix F. Validation of Telepresence_2Systems

Two acceptance and two rejection test cases, mostly adapted from the test cases in appendix D.II, are applied to the *Telepresence_2Systems* specification. We again make use of the testing capacities of LOLA. Although they would be necessary in a real-life validation, simulation and verification are not to be used here to simplify the approach.

First Acceptance Test Case

This process tests whether the possible *Report*s are *Denied* or *TimeOut* when the responder's door is closed (by default in the specification). System1 is the initiator and system 2 is the responder.

```
process AcceptTest1[Contact1, Report1, Success]: noexit :=
(* Test for denied or timeout report when the door is closed (default) *)
   Contact1;
   (
      Report1 ! Denied; Success; stop
   []
      Report1 ! TimeOut; Success; stop
   )
endproc (* AcceptTest1 *)
```

Specification 17: Acceptance test case 1 for Telepresence_2Systems

We use the command *TestExpand* to test all possible scenarios. The result, expressed below, shows our test case to be a **must pass** test again. This result suggests that the specification might conform to the *Telepresence* specification.

```
lola> TestExpand -1 Success AcceptTest1 -i

 Composing behaviour and test :

    Analysed states       = 370
    Generated transitions = 506
    Duplicated states     = 0
    Deadlocks             = 0

    Process Test = accepttest1
    Test result  = MUST PASS.

                   successes = 137
                       stops = 0
                       exits = 0
               cuts by depth = 0
```

First Rejection Test Case

This second process tests that *Report* cannot output the value *Success* when the responder's door is closed (by default in the specification). System1 is the initiator and system 2 is the responder.

```
process RejectTest1[Contact1, Report1, Fail]: noexit :=
(* Test for denied or timeout report when the door is closed (default) *)
   Contact1;
   Report1 ! Success; Fail; stop
endproc (* RejectTest1 *)
```

Specification 18: Rejection test case 1 for Telepresence_2Systems

The *TestExpand* procedure returns **reject**, meaning that our specification behaves properly in that matter, just as *Telepresence* did before.

```
lola> TestExpand -1 Fail RejectTest1 -i

 Composing behaviour and test :

    Analysed states       = 233
    Generated transitions = 232
    Duplicated states     = 0
    Deadlocks             = 67

    Process Test = rejecttest1
    Test result  = REJECT.

                   successes = 0
                       stops = 67
                       exits = 0
               cuts by depth = 0
```

Second Acceptance Test Case

This more complex process tests that, when the responder *Open*s his/her door and the initiator *Contact*s the responder, the *Report* should output a *TimeOut* or possibly a *Success* when the initiator *Terminate*s the connection. In this example, system 2 is the initiator and system 1 is the responder (but this has no repercussion because systems 1 and 2 are perfectly symmetrical).

```
process AcceptTest2[Contact2, Open1, Terminate2, Report2, Success]: noexit :=
(* Test for success (or timeout) report when the responder opens the door *)
   Open1;
   Contact2;
   (
      Report2 ! TimeOut; Success; stop
   []
      Terminate2;
      (
         Report2 ! Success; Success; stop
         []
         Report2 ! TimeOut; Success; stop
      )
   )
endproc (* AcceptTest2 *)
```

Specification 19: Acceptance test case 2 for Telepresence_2Systems

Since the transition system generated from the *Telepresence* specification is infinite when there is a transmission, we must again limit the depth of the search (here to 15 events) of the *TestExpand* procedure. This generates an extremely large number of executions (862999) that should give us confidence in the result.

```
lola> TestExpand 15 Success AcceptTest2 -i -y

 Composing behaviour and test :

    Analysed states       = 521796
    Generated transitions = 1384734
    Duplicated states     = 0
    Deadlocks             = 60

    Process Test = accepttest2
    Test result  = MAY PASS.

    862999 executions analysed:

                   successes = 76919
                       stops = 60
                       exits = 0
               cuts by depth = 786020
```

LOLA outputs **may pass** as a result. On 862999 executions, 76919 were successful, 60 were unsuccessful, and the remaining 786020 were unfinished due to the search depth of 15. We could use here the option `-s` of *TestExpand* to get all 60 traces leading to deadlocks, in the form of a monolithic LOTOS tree. Then, we could compare these traces with the ones found in the previous *Telepresence* specification to verify that no new problem was introduced.

Second Rejection Test Case

This last test was not in the previous test suite (appendix D.II). We create it here in order to ensure that we have no additional error related to invalid scenarios due to communication between systems (as presented in section 6.4.1). *RejectTest2* tests that when an initiator (system 1) *Contact*s the responder (system 2), the latter does not get any *Report* of any kind.

```
process RejectTest2[Contact1, Report2, Fail]: noexit :=
(* Test for denied or timeout report when the door is closed (default) *)
   Contact1;
   Report2 ? Anything: Tag; Fail; stop
endproc (* RejectTest2 *)
```

Specification 20: Rejection test case 2 for Telepresence_2Systems

The *TestExpand* procedure returns **reject**, meaning that our specification does not add to the problems we already encountered. Hence, this is another step in the conformance testing of *Telepresence_2Systems* w.r.t. *Telepresence* and the requirements.

```
lola> TestExpand -1 Fail RejectTest2 -i

 Composing behaviour and test :

    Analysed states       = 233
    Generated transitions = 232
    Duplicated states     = 0
    Deadlocks             = 67

    Process Test = rejecttest2
    Test result  = REJECT.

                   successes = 0
                       stops = 67
                       exits = 0
               cuts by depth = 0
```

List of Figures

List of Specifications

Index