LOTOS Interpretation of Timethreads: A Method and a Case Study

F. Bordeleau, D. Amyot

Report SCE-93-34
December 8, 1993

Department of Systems and Computer Engineering
Carleton University
Ottawa Canada K1S 5B6
email: francis@sce.carleton.ca
damyot@csi.uottawa.ca

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

LOTOS Interpretation of Timethreads: A Method and a Case Study

Francis Bordeleau[1]
Daniel Amyot[2]

[1] *Real-Time and Distributed Systems Research Group*
Carleton University
Department of Systems and Computer Engineering
Ottawa, Ont., Canada K1S 5B6
email: francis@sce.carleton.ca

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

Abstract

Timethreads are a new notation for visual description of path behavior. Also, a design process based on timethreads, namely timethread-centered design process, has been defined. In this report, we discuss the integration of the FDT LOTOS in the timethread-centered design process. The objective of such an integration is to provide formal support for timethread transformations with LOTOS. For this purpose, we first define a LOTOS interpretation method for timethreads. The method allows the generation of LOTOS specifications from timethread diagrams. Then, we show how the LOTOS interpretation method for timethreads applies in practice by conducting a case study. This case study also serves to identify topics for future research in relation with the definition of a formal framework to support the timethread-centered design process.

1. Introduction

1.1 Overview

Timethreads have been defined in [Buh 93 & BuC 93] as a new visual notation for path description of distributed systems. Timethreads visually illustrate causality sequences of activities through systems. Also, in [BuC 93 & BCP 93], a design process based on timethreads, namely timethread-centered design process, is defined. Although timethreads have only existed for a few years, the notation has been taught by Prof. Buhr to many students at Carleton University and to

hundreds of engineers in industries, in both Canada and USA, through a series of courses. The concept of timethreads has been very well accepted by both students and engineers mainly because it is based on a very natural way of thinking when designing real-time and distributed systems. The notation is easy to understand and it facilitates both description and visualization of system path behaviour.

The formal aspect of timethreads has been partially discussed in relation with both Petri nets in [FCB 93] and LOTOS in [Vig92] (timethreads were called "slices" in [Vig92]). However, timethreads do not have, at this point in time, a complete formal semantics. One direct consequence of this is that timethread manipulations can only be conducted informally. This means that a timethread diagram *D*', which has been obtained from a diagram *D* by successive timethread manipulations, can not be validated with respect to the former diagram *D* in a formal way. Thus, we can not verify that *D*' is correct with respect to *D*. Then, since timethread manipulations play a key role in the timethread-centered design process, we need to define, in order to use a timethread-centered design process in an industrial environment, a formal framework for timethreads that will enable the support of timethread manipulations. Also, the definition of such a framework would constitutes an important step towards the definition of a tool to support the timethread-centered design process.

This report, together with [Bor 93] and [Amy 93], constitutes the starting point of a new project, called **FIT** (**F**ormal Method **I**ntegration in the **T**imethread-Centered Design Process), which aims at defining a formal framework to support the timethread-centered design process. In this report, we discuss the integration of the FDT LOTOS in the timethread-centered design process. The objective of such an integration is to provide formal support for timethread manipulations using LOTOS. For this purpose, we think that if LOTOS specifications can be obtained from timethreads diagrams, then CPTs (Correctness Preserving Transformations) defined for LOTOS may be adapted to provide such support.

1.2 Objectives

In order to integrate LOTOS in the timethread-centered design process, the first step consists in defining an interpretation method that allows the generation of LOTOS specifications from timethread diagrams. In this way, the objectives of this report are:

1°) to define a LOTOS interpretation method for timethreads,

2°) to conduct a case study illustrating how the interpretation method applies in practice.

The LOTOS interpretation method for timethreads defined in this report is based on the work described in [Bor 93] and [Amy 93]. The interpretation method is a general one that allows the generation of LOTOS specifications from timethread diagrams. For the purpose of the case study, we use the traveler system described in [BuC 93]. We show how a LOTOS specification can be derived from the timethread diagram of the traveler system using the LOTOS interpretation method for timethreads defined in this report. The case study also serves to identify topics for future research in relation with the definition of a formal framework to support the timethread-centered design process.

1.3 Organization

The report is organized as follows. In section 2, we define the LOTOS interpretation method for timethreads. In this section, we show how the concept of generic LOTOS interpretation methods defined in [Bor 93] may be adapted for the definition of a specific LOTOS interpretation method for timethreads. We also show how the LOTOS semantics for timethreads, defined in [Amy 93], applies in the definition of the LOTOS interpretation method for timethreads. In section 3, we illustrate how this interpretation method may apply in practice to generate LOTOS specifications from timethread diagrams. For this purpose the traveler system is used as a case study. This case study will serve to identify future research topics for the FIT project. Section 4 presents a few executions of the traveler specification, to get an idea of what information is contained in such a specification. Finally, in section 5, we discuss the conclusions of this report and draw up a list of topics for future research.

2. A LOTOS Interpretation Method for Timethreads

In [Bor 93], the concept of formal interpretation method is defined. A formal interpretation method allows the interpretation of a given design in terms of a given formal semantic model. In [Bor 93], the concept of formal interpretation method is defined in relation with component-centered type of designs, which corresponds to the conventional design process in which components are decomposed into interacting subcomponents until every bottom level component is simple enough to be considered as a primitive component [Tur 93 & Buh 93a].

In this report, we apply the concept of interpretation method for the LOTOS interpretation of timethread diagrams. This LOTOS interpretation method for timethreads is illustrated in figure 1.

In the following subsections, we describe each part of the LOTOS interpretation method for timethreads: the timethread decomposition method (§2.1), the LARG model (§2.2), the LAEG method (§2.3), and the composition of the complete specification method (§2.4).

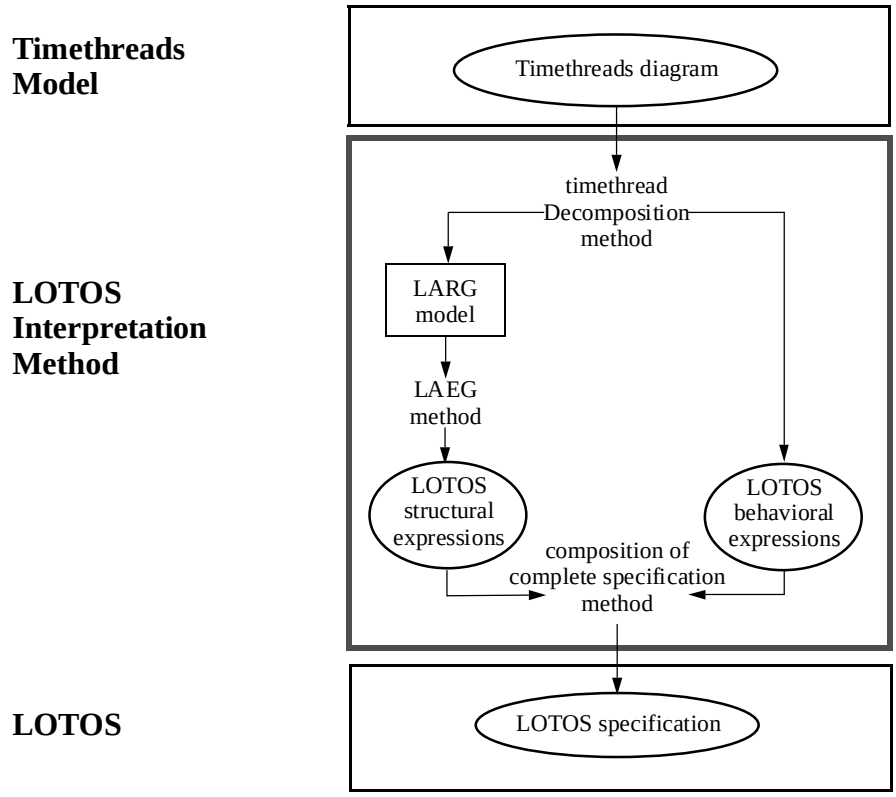


Figure 1: LOTOS interpretation method for Timethreads

2.1 Timethread decomposition method

The timethread decomposition method is based on the work described in [Amy 93], where the LOTOS semantics of both the different types of individual timethreads and the different type of timethread interactions is discussed. As a result, skeletons of LOTOS process corresponding to the different types of timethreads and LOTOS structural expressions corresponding to the different possible types of single interactions are given. Also, different issues in relation with the LOTOS interpretation of timethreads have been raised.

The LOTOS interpretation method for timethreads consists of two steps:

- Mapping of the timethreads diagram onto a LARG.
- Mapping of the path behaviour of individual timethreads onto LOTOS behavioural expressions.

Figure 2 shows an example where a small diagram with two timethreads (***P*** and ***Q***), representing a system "design", is mapped onto a LARG with two processes. The timethreads interact on the event *GoQ*, and this is reflected in the LARG by a 2-way rendez-vous. Usually, we obtain one LOTOS process for each timethread. However, in some cases, like synchronized segments of timethreads, we need to introduce additional LOTOS processes. In figure 3, the LOTOS behavioural expressions corresponding to timethreads ***P*** and ***Q*** is given..

The final specification would be a simple combination of the structural expression and the behavioural expressions. The traveler example of section 3 will develop these issues more deeply.

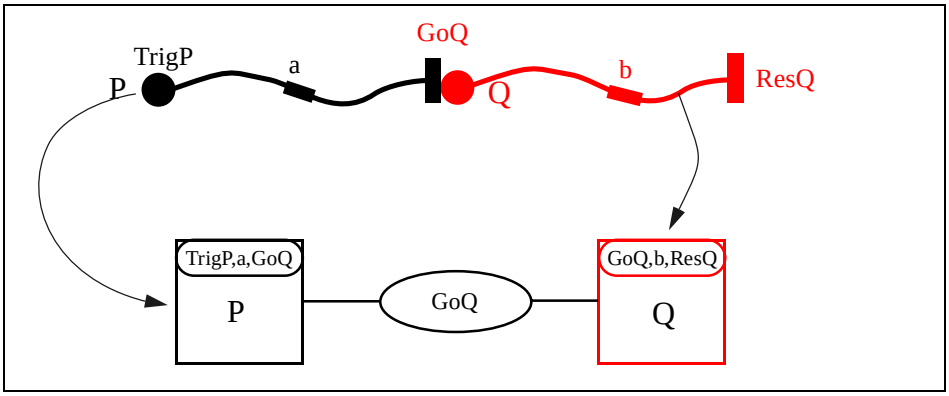


Figure 2: Example of a timethread diagram and its corresponding LARG

```
behaviour

  P [TrigP, a, GoQ]
  |[GoQ]|
  Q [GoQ, b, ResQ]
```

```
process P [TrigP, a, GoQ] : noexit :=
    TrigP; (a; GoQ; stop ||| P [TrigP, a, GoQ])
endproc

process Q [GoQ, b, ResQ] : noexit :=
    GoQ; (b; ResQ; stop ||| Q [GoQ, b, ResQ])
endproc
```

Figure 3: Structure part and LOTOS processes corresponding to the LARG

2.2 The LARG model

The LARG model has been developed to serve as the intermediate structural model in the LOTOS interpretation method. An example of a LARG, in which the different types of LARG components are identified, is illustrated in figure 4. The LARG model possesses only one type of structural component, called *process*, and one type of interaction which is rendez-vous interaction. Interactions between processes are realized by means of synchronization on *gates*. The LARG model allows the representation of N-way interactions (for $N \geq 1$) and possesses a high level interaction operator that corresponds to the LOTOS parallel operator. Also, because the generation of LOTOS structural expressions must be allowed from LARGs, the LARG model has been developed in

such a way that the LARG artifacts, i.e. processes and gates, can directly be mapped to LOTOS structural constructs, i.e. LOTOS processes and LOTOS gates. Finally, for the purpose of the LAEG 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].

In the timethread interpretation method, each timethread of the timethread diagram is mapped onto a LARG process and interactions between timethreads are mapped onto interaction gates. Therefore, the term *structure* refers to timethread structure, i.e. the topology of interacting timethreads.

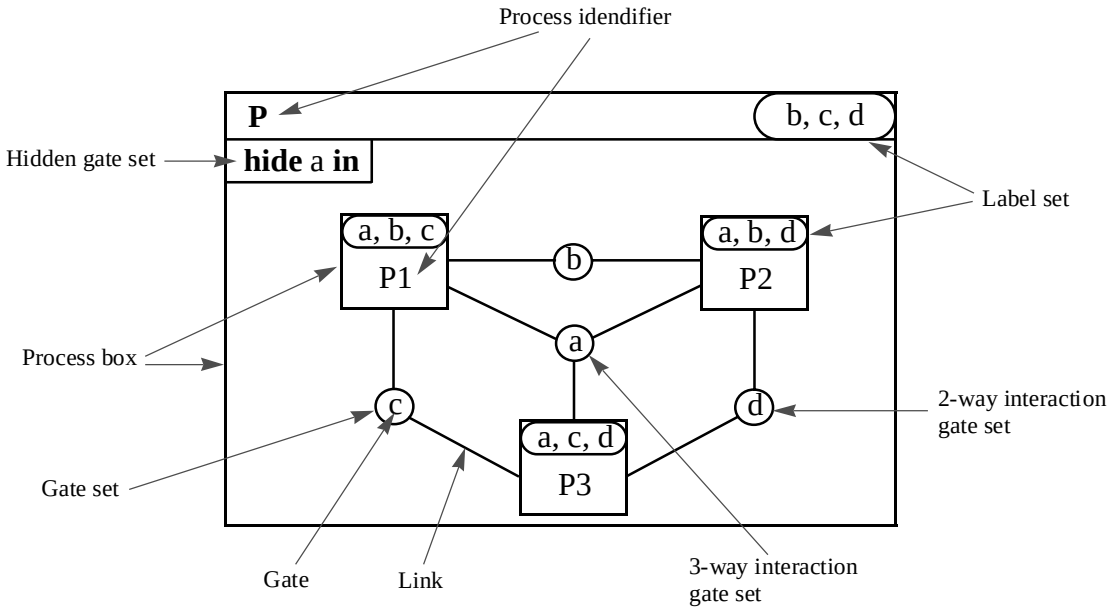


Figure 4: Example of a LARG

2.3 The LAEG method

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

1°) LARG analysis, and

2°) generation of LOTOS structural expressions.

2.3.1 LARG Analysis

In the case of timethreads interpretation, the LARG analysis phase is reduce to non-determinism identification. Also, the only type of non-determinism allowed in timethreads is non-deterministic interaction-choice.

We say that a gate *g* is the source of non-deterministic interaction-choice in a LARG *P*, iff:

1°) *g* is contained in more than one gate set (GS) in *P*, and

2°) every GS containing *g* is linked on one side to a constant set of processes, called the *root process set* of the non-deterministic interaction-choice, 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 non-deterministic interaction-choice.

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 this definition, *g* can be of any types of interactions, i.e. N-way interaction for any N > 1.

In figure 5, an example of a non-deterministic interaction-choice LARG is given. In this LARG, gate *a* is the non-deterministic interaction-choice 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*.

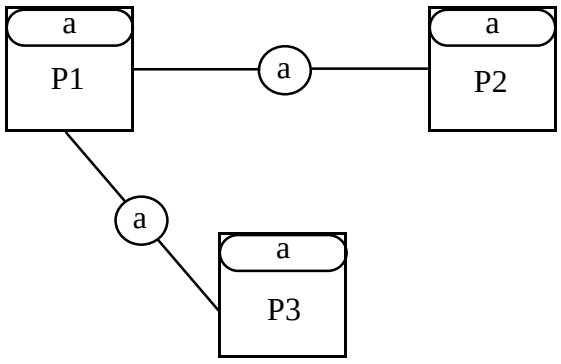


Figure 5: Non-deterministic interaction-choice LARG

2.3.2 Generation of LOTOS structural expressions

The second phase consists in generating LOTOS structural expressions from LARGs. This phase is essential since LOTOS only possesses binary operators. 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 6. Figure 6(b) gives an equivalent binary grouping LARG which has been obtained by successive applications of the grouping algorithm. The grouping sequence used in figure 6 has been arbitrarily chosen, and is only one of many possible ones.

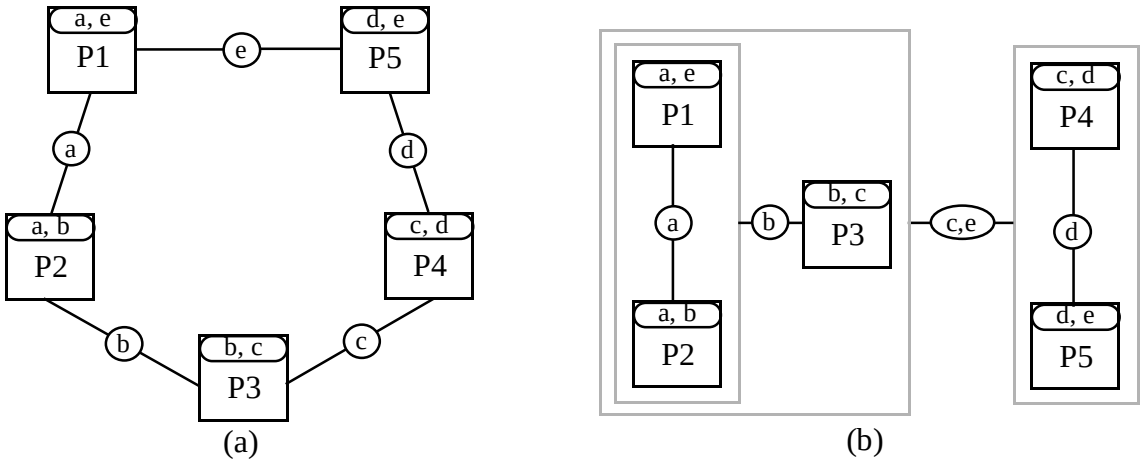
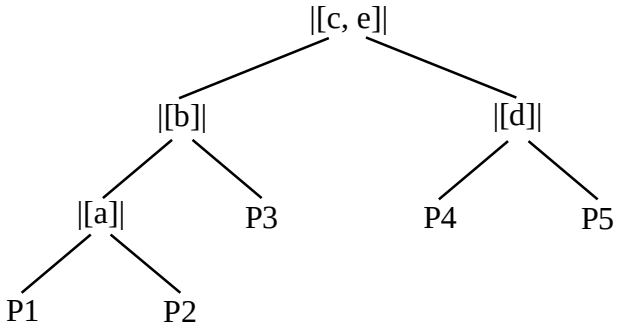


Figure 6: Binary grouping of a LARG

The tree representation of the LARG of figure 6(b) and its associated LOTOS structural expression are given in figure 7. 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])

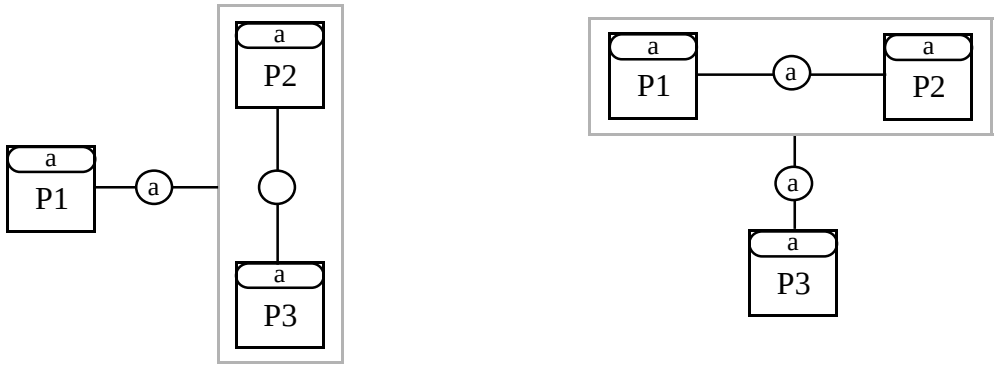
Figure 7: Tree representation and LOTOS architectural expression of the linearized LARG

Non-deterministic interaction-choice LARG

Groupings in non-deterministic LARGs is more problematic because, in such cases, some groupings violate the interaction semantics of the LARG. For example figure 8(a) and 8(b) represent two different groupings of the LARG of figure 5. 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 8(a) corresponds to a correct interpretation of figure 5, while figure 8(b) corresponds to an incorrect one.

To eliminate non-determinism from non-deterministic interaction-choice LARGs, a technique called *non-deterministic interaction-choice grouping* is defined in [Bor 93]. In non-deterministic

interaction-choice grouping, we group together all choice processes, i.e., all processes contained in the choice process set (see [Bor 93] for more details on grouping techniques defined to eliminate non-determinism in LARG). Figure 8(a) illustrates an example of the application of the non-deterministic interaction-choice grouping technique.



(a)

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

(b)

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

Figure 8: 2-way non-deterministic parallel-interactions choice grouping

2.4 The 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 diagram, and the different LOTOS behavioral expressions, each of which expresses the activity sequence in a single timethread, in a global LOTOS specification. The resulting global LOTOS specification reflects the path behaviour of the complete timethread diagram.

3. Case Study

The traveler system, shown in figure 10, is not a truly modern 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.

3.1 Informal description of the traveler system

Travelers use a *traveler system* to get to a certain destination. The timethread diagram of figure 9 shows a "*use case*" delimiting the system (black box) and its environment. To transform this black box into a gray box showing how a traveler gets to its destination, we need a more complete description.

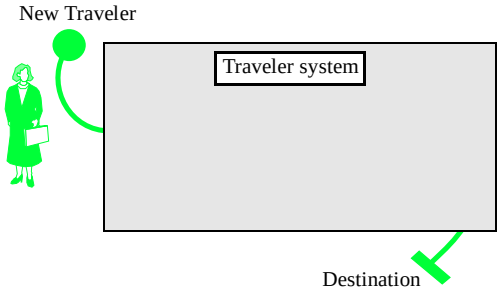


Figure 9: Use case of the traveler system

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, with corresponding activities (in fig. 10) between parenthesis. When a new traveler comes (*Tnew*), he/she phones the dispatcher for a cab (*TphoneD*), goes to a rendez-vous point, gets in the cab (*TgetinC*), has a taxi ride (*TCride*), gets out the cab (*TgetoutC*), and goes to the airport (*Tairport*). Then, he/she waits for a plane, gets on the plane (*TgetonP*), has a flight 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 rendez-vous 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*).

3.2 Timethread diagram

Following the complete description of the last section, the simple use case presented in figure 9 can be refined, using a timethread-centered design process [BuC 93], into a detailed path description: the timethread diagram of figure 10.

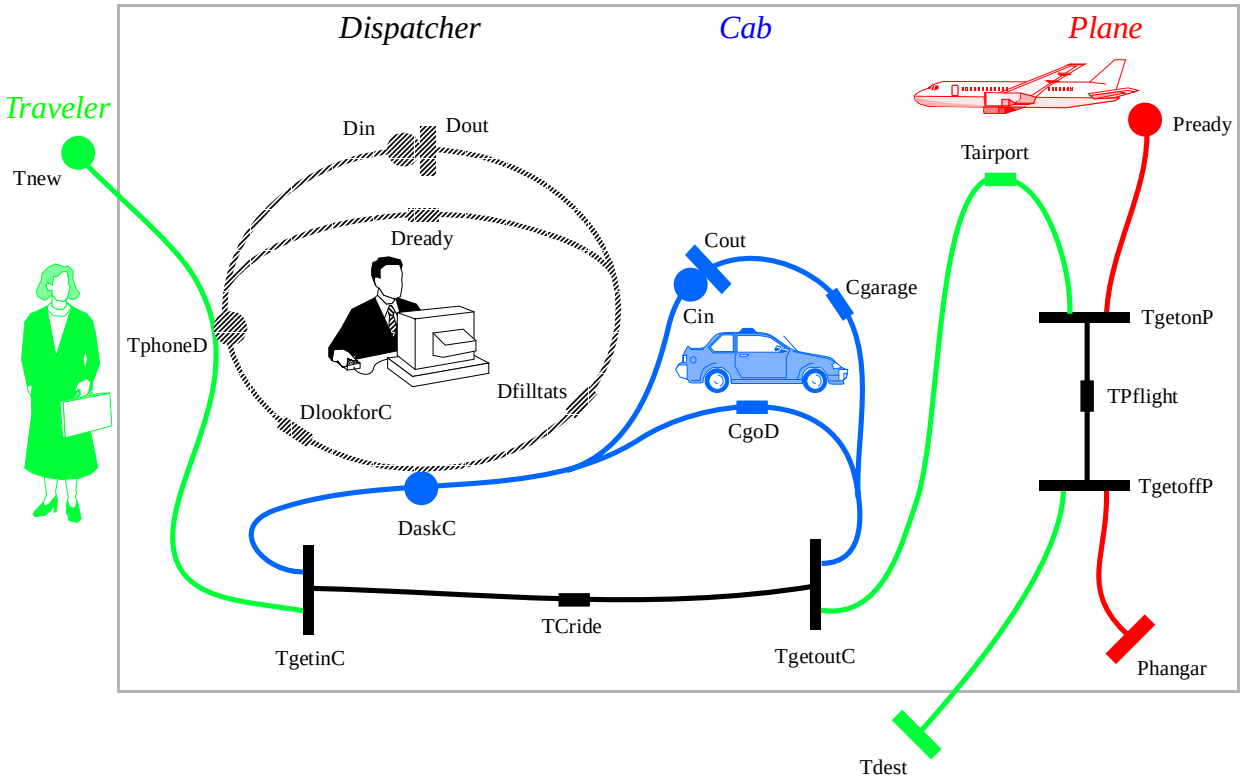


Figure 10: Timethread diagram of the traveler system

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 9). Every activities in the SUD will be “hidden” from a LOTOS point of view.
- A timethread is neither a component, an agent, nor an object, as the diagram could suggest. Timethreads *span* through 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 patterns 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. Patterns, colours, and identifiers can be used for this purpose.

3.3 LARG representation

The first step for the obtention of a LOTOS specification from a timethread diagram is to map the latter on a LARG (fig. 11).

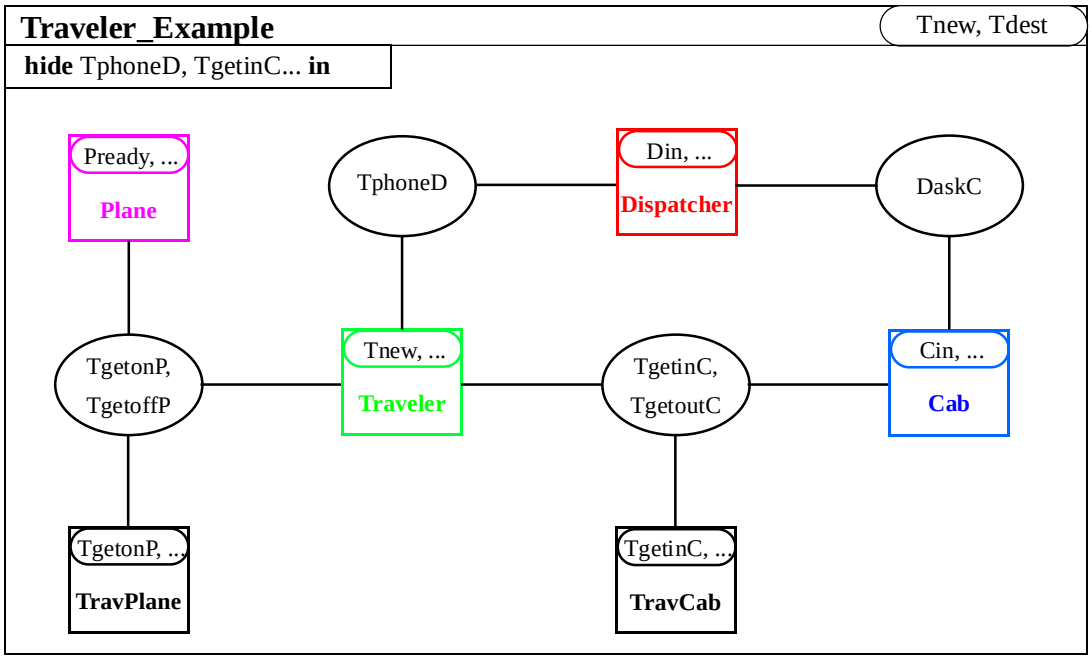


Figure 11: LARG representation of the traveler system

The ***hide*** part of process ***Traveler_Example*** hides labels that are in the SUD, i.e., every labels except *Tnew* and *Tdest*, which are external events. Internal processes also hide their internal activities, e.g., process ***Dispatcher*** hides *DlookforC*, *Dfillstats* and *Dready*[1]. The hiding of activities on a timethread diagram is still an open issue. LOTOS provides much flexibility on this aspect, but conventions for the mapping are still needed. In the traveler example, there are only two external events (*Tnew* and *Tdest*), and actions (internal activities) of a timethread are hidden by default within the corresponding LOTOS process.

Figure 11 shows one LOTOS process for each timethread, plus two extra processes (***TravPlane*** and ***TravCab***). These processes are needed for modularity reasons. They can ease transformations like regrouping and splitting [Amy 93], and they express more clearly common actions between synchronized timethreads.

The next step of the transformations concerns the binary grouping of internal processes, which allows a direct mapping onto a LOTOS structure of processes. The grouping is done using the

1. The ***hide*** parts of internal processes are not shown in de LARG for space reason, but we assume they are there.

LAEG method [Bor 93]. Many different groupings can result from this algorithm, and the final choice should not be arbitrary. Design decisions such as performance and location of components and/or processes should tell us which grouping is the best. However, no such metrics have been defined yet. Figure 12 shows one possible grouping.

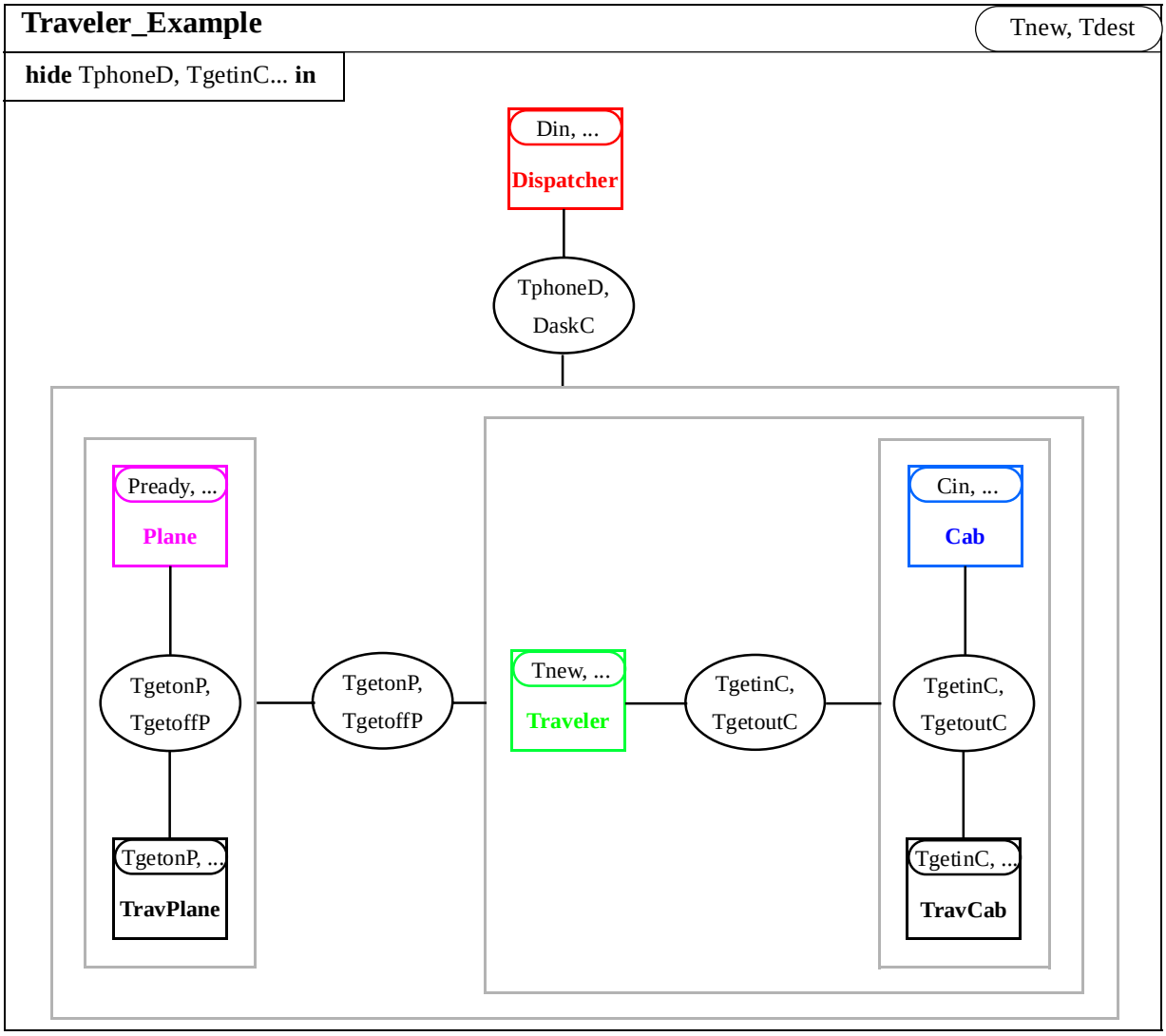


Figure 12: Binary grouping of the traveler system

From this LARG, we can derive the structural section (unfortunately called *behaviour* section in LOTOS) of the LOTOS specification (see [Bor 93] for more details):

```
specification Traveler_Example[Tnew  (* New traveler wants to travel *),
                               Tdest (* Traveler arrives to destination *) ] : noexit

behaviour (* Architecture obtained from the LARG *)

hide  (* hidden interactions *)
    TphoneD,        (* Traveler phones Dispatcher for a cab *)
    TgetinC,        (* Traveler gets in the cab *)
    TgetoutC,       (* Traveler gets out the cab *)
    TgetonP,        (* Traveler gets on the plane *)
    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

    Dispatcher[Din, TphoneD, DaskC, Dout]
    |[TphoneD, DaskC]|
    (
        (
            (
                Cab[Cin, DaskC, TgetinC, TgetoutC, Cout]
                |[TgetinC, TgetoutC]|
                TravCab[TgetinC, TgetoutC]
            )
            |[TgetinC, TgetoutC]|
            Traveler[Tnew, TphoneD, TgetinC, TgetoutC, TgetonP, TgetoffP, Tdest]
        )
        |[TgetonP, TgetoffP]|
        (
            Plane[Pready, TgetonP, TgetoffP, Phangar]
            |[TgetonP, TgetoffP]|
            TravPlane[TgetonP, TgetoffP]
        )
    )
```

3.4 Development of a timethread

Once the structure of the LOTOS specification is defined, every process has to be filled with its behaviour (or path description). For instance, figure 11 focuses on the timethread ***Traveler*** and its corresponding complete representation extracted from the LARG. Again, events are considered as LOTOS gates and the activity *Tairport* is hidden (so it is an internal action). Note that actions *TCride* and *TPflight* are not included in the label set since they are considered as internal activities of processes ***TravCab*** and ***TravPlane***, respectively.

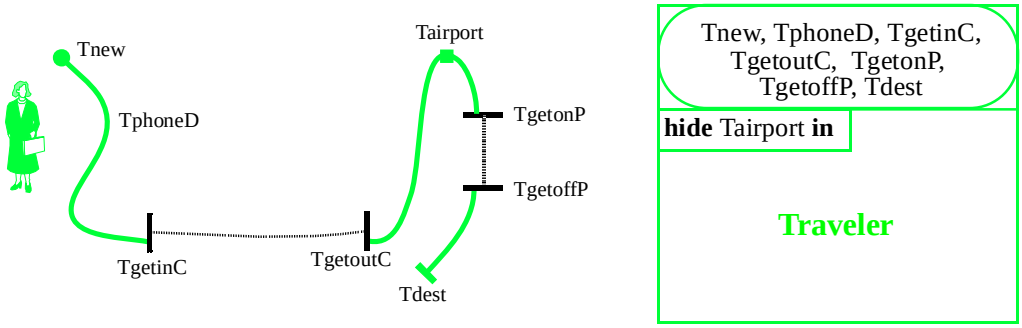


Figure 13: Timethread *Traveler* and its corresponding LARG process

The LOTOS interpretation of this timethread is obtained using the semantics introduced in [Amy 93]. Different levels of abstraction could be used here. In this LOTOS process, the event *TphoneD* represents an asynchronous interaction (in passing), interpreted as the interleaving sub-process *TphoneD; **stop***. Since LOTOS allows synchronous interactions only, we have to simulate asynchronous interactions in this way.

```
(* Timethread Traveler *)
    process Traveler[Tnew, TphoneD, TgetinC, TgetoutC, TgetonP, TgetoffP, Tdest] : noexit :=
        hide Tairport in  (* hidden action *)
        Tnew;
        (
            TphoneD; stop  (* in passing interaction *)
            |||
            (
                TgetinC;
                TgetoutC;
                Tairport;
                TgetonP;
                TgetoffP;
                Tdest; stop
            )
            |||
            (* recursive call *)
            Traveler[Tnew, TphoneD, TgetinC, TgetoutC, TgetonP, TgetoffP, Tdest]
        )
    endproc (* Traveler *)
```

The other 5 processes are also mapped onto LOTOS to form the final specification of our system (presented in the Appendix A).

4. Simulation and validation

Such specification will be helpful, in later stages of the design, as a formal support to timethread transformations, which are still research issues. However, this does not mean that this type of specification is not useful as it is, in the contrary. We can “execute” the specification, with common LOTOS tools, either to get the feeling that our diagram corresponds in some way to the func-

tionality defined in the requirements, or to detect possible problems which will have to be solved during later stages of the design. This simulation can effectively, at some level, leads to some questions that the designer will have to answer with some refinement.

Two similar tools were used for the step-by-step simulation of the ***Traveler_Example*** specification. The first one is XELUDO (Environnement LOTOS de l'Université d'Ottawa), on X-Windows SUN workstations, and its TTY version (for VT-100 terminal) ELUDO. The second tool was the PC version of LOLA (LOtos LAboratoty) from the University of Madrid.

Different levels of specification were used for the simulation. In the level 1 specification (without recursion), all recursive calls were removed. Therefore, only one instance of each process was allowed. For instance, the process ***Traveler*** becomes:

```
(* Timethread Traveler *)
    process Traveler[Tnew, TphoneD, TgetinC, TgetoutC, TgetonP, TgetoffP, Tdest] : noexit :=
        hide Tairport in  (* hidden action *)
        Tnew;
       (
            TphoneD; stop  (* in passing *)
            |||
            (
                TgetinC;
                TgetoutC;
                Tairport;
                TgetonP;
                TgetoffP;
                Tdest; stop
            )
        )                  (* No more recursion! *)
    endproc (* Traveler *)
```

The simulation of this specification was straightforward. No unexpected problem was detected. The Appendix B shows an instance of a trace obtained with ELUDO (I) and another trace obtained with the help of LOLA (II). Level 1 specifications are useful only in the early stages of the design process, when a fast simulation is needed to check a few simple properties or to get the feeling our timethread diagram is right.

A level 3 specification (like the one in Appendix A) is more useful for thinking about the design. Appendix B (III) presents a trace (obtained with LOLA) where the resulting event *Tdest* is reached. However, although knowing that the purpose of our system can be fulfilled is essential, it is very interesting to look for possible problems. This is in fact the goal of testing. Step-by-step simulation can help us test our specification with different scenarios, in order to observe the system's reactions.

Part IV of Appendix B presents a case where the dispatcher, after receiving two requests from two travelers, finally finds a taxi. The designer could wonder what was his initial intention there: how many requests can the dispatcher accumulate before he tells the travelers he cannot take any more requests? Is there any mean for the dispatcher to tell the next travelers that they would have to call

back later, when the system permits it? In the first loop of timethread ***Dispatcher***, is it normal that the dispatcher fills his statistics 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 happens with travelers and planes? All these questions could be raised only 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.

Part V presents a problem about the number of instances. It appears that a unbounded number of travelers can request a taxi. Again, how many requests can the dispatcher can deal with? If the dispatcher is not in the office, is there a way to signal the travelers that the system may not work properly? Also, when many requests come to the dispatcher, should he deal with them in some order, e.g., first in first out?

Appendix B (VI) also shows a short simulation using ELUDO and a mixed-level specification. There are a maximum of three travelers (level 2 process, with a bounded number of instances), only one dispatcher (level 1), a maximum of 2 cabs (level 2) and an unbounded number of planes (level 3). Mixed-level specifications allow more control on the number of instances of timethreads, resulting in more realistic simulations [Amy 93]. In this example, we can see that two travelers (we can call them *T1* and *T2*) got into two different taxis (*C1* and *C2*). After their rides, they have to get out their respective cabs. However, they simulation shows at this point four possible actions. This is an interaction problem where *T1* can get out of *C1* or *C2*, and *T2* can also get out either of *T1* or *T2*, explaining the four different choices! The designer knows that his system will have to resolve some concurrency problem (like this interaction) later on.

Verification could be done using, for example, temporal logic over the symbolic extension of a LOTOS specification [Ghr 92]. This extension allows the verification of all possible traces in the transition system corresponding to the specification. SELA, a tool integrated in XELUDO, gives the full expansion of a LOTOS process in the form of a tree, or more precisely a transition system. Appendix B (VI) gives, as a short example, the extension of process ***Traveler*** in a level 1 specification of the traveler system. LOLA also possesses such an extension function, but it uses a very different format.

Other testing tools using goal-oriented execution, trace theory and temporal-logic could be integrated into a timethreads-LOTOS simulation/testing environment.

5. Conclusions

In this report, we defined a LOTOS interpretation method for timethreads. The definition of this method is based on the work described in [Bor 93] and [Amy 93]. This method allows the generation of LOTOS specifications from timethread diagrams. The definition of such a method constitutes the first phase of the FIT project which aims at defining a framework for the integration of formal methods in the timethread-centered design process. Also, we showed through the case study that the interpretation method can be applied in practice, and that the specification we obtain can be executed as a fast-prototype in the early stages of the design process.

The use of the method with more complex examples requires the definition of a more rigorous timethread interpretation method which would enable the generation of LOTOS behavioral expressions, or LOTOS processes, from any arbitrarily timethreads. For this purpose, we need to define a more general LOTOS semantics for timethreads that would allow the interpretation of individual timethreads as the composition of timethreads constructors. This is part of ongoing research.

In order to allow the use of the LOTOS interpretation method, defined in this report, in a global framework for the integration of LOTOS in the timethread-centered design process, the complete formalization of the method is now required. The complete formalization of the LOTOS interpretation method will first require the definition of a formal representation of timethreads notations, possibly in a BNF. This will then enable the formal definition of the timethread interpretation method, which is responsible for both the generation of behavioral expressions for individual timethreads and the generation of a LARG from a timethread diagram.

5.1 Future Research

- Define in a complete formal way the LOTOS interpretation method for timethreads, in particular the timethreads interpretation method defined in [Amy93] need to be completely formalized,
- Develop other case studies using more complex examples, e.g. the elevator system and the MTU system [Buh 93a],
- Define correctness preserving transformations (CPTs) for timethreads based on LOTOS CPTs,
- Define a correspondence, based on LOTOS, between a timethread diagram and a skeleton architecture,
- Define similar interpretation methods for other formal semantic model, e.g. Petri nets and event structures

6. Acknowledgments

Many people, from both Carleton University and University of Ottawa, contributed to this report. We are particularly grateful to Professor R.J.A. Buhr and Professor L. Logrippo for all the constructive discussions, comments and encouragements. Also, many thanks to Ron Casselman and Professor Abdellatif Obaid for their helpful comments. Finally, Jacques Sincennes and Jean Tourrilhes have helped a lot with the LOTOS toolkit XELUDO.

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

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)

[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)

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

[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)

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

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

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

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

[Ghr 92] B. Ghribi, "A Model Checker for LOTOS", Master's thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada (1992)

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

[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* (1990), North-Holland, 229-242

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

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

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

[Vig 92] M. Vigder, "Integrating Formal Techniques into the Design of Concurrent Systems", Ph.D. Thesis OCIEE-92-03, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1992)

Appendix A: Final LOTOS specification

```
(* Traveler example; Daniel Amyot, December 1993 *)
(* Level 3 specification *)

specification Traveler_Example[Tnew  (* New traveler wants to travel *),
                               Tdest (* Traveler arrives to destination *) ] : noexit

behaviour (* Architecture obtained from the LARG *)

hide  (* hidden interactions *)
    TphoneD,        (* Traveler phones Dispatcher for a cab *)
    TgetinC,        (* Traveler gets in the cab *)
    TgetoutC,       (* Traveler gets out the cab *)
    TgetonP,        (* Traveler gets on the plane *)
    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

    Dispatcher[Din, TphoneD, DaskC, Dout]
    |[TphoneD, DaskC]|
    (
        (
            (
                Cab[Cin, DaskC, TgetinC, TgetoutC, Cout]
                |[TgetinC, TgetoutC]|
                TravCab[TgetinC, TgetoutC]
            )
            |[TgetinC, TgetoutC]|
            Traveler[Tnew, TphoneD, TgetinC, TgetoutC, TgetonP, TgetoffP, Tdest]
        )
        |[TgetonP, TgetoffP]|
        (
            Plane[Pready, TgetonP, TgetoffP, Phangar]
            |[TgetonP, TgetoffP]|
            TravPlane[TgetonP, TgetoffP]
        )
    )

where

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

(* Timethread Traveler *)
    process Traveler[Tnew, TphoneD, TgetinC, TgetoutC, TgetonP, TgetoffP, Tdest] : noexit :=
        hide Tairport in  (* hidden action *)
        Tnew;
       (
            TphoneD; stop  (* in passing *)
            |||
```

```
            (
                TgetinC;
                TgetoutC;
                Tairport;
                TgetonP;
                TgetoffP;
                Tdest; stop
            )
            |||
            (* recursive call *)
            Traveler[Tnew, TphoneD, TgetinC, TgetoutC, TgetonP, TgetoffP, Tdest]
        )
    endproc (* Traveler *)

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

(* Timethread Dispatcher *)
    process Dispatcher[Din, TphoneD, DaskC, Dout] : noexit :=
        hide Sync in (* Constrained start *)
            Dis1[Din, Sync] |[Sync]| Dis2[Sync, TphoneD, DaskC, Dout]
        where
            process Dis1[Din, Sync] : noexit :=
                Din; (Sync; stop ||| Dis1[Din, Sync])
            endproc (* Dis1 *)

            process Dis2[Sync, 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
                    Sync; DisLoop[Sync, TphoneD, DlookforC, DaskC, Dfillstats, Dready, Dout]
                where

                    (* Loop part of the timethread Dispatcher *)
                    process DisLoop[Sync, TphoneD, DlookforC, DaskC, Dfillstats, Dready, Dout]
                                    : noexit :=
                        TphoneD;
                        DlookforC;
                        (
                            DaskC; stop  (* in passing *)
                            |||
                            Dfillstats;
                            (
                                Dready; DisLoop[Sync, TphoneD, DlookforC, DaskC, Dfillstats,
                                                Dready, Dout]
                                []
                                Dout; Dis2[Sync, TphoneD, DaskC, Dout]
                            )
                        )
                    endproc (* DisLoop *)

            endproc (* Dis2 *)
    endproc (* Dispatcher *)

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

```
(* Timethread Cab *)
    process Cab[Cin, DaskC, TgetinC, TgetoutC, Cout] : noexit :=
        hide Sync in (* Constrained start *)
            Cab1[Cin, Sync] |[Sync]| Cab2[Sync, DaskC, TgetinC, TgetoutC, Cout]
        where
            process Cab1[Cin, Sync] : noexit :=
                Cin; (Sync; stop ||| Cab1[Cin, Sync])
            endproc (* Cab1 *)

            process Cab2[Sync, DaskC, TgetinC, TgetoutC, Cout] : noexit :=
                (* hidden actions *)
                hide
                    CgoD,    (* Cab goes to wait the dispatcher *)
                    Cgarage  (* Cab goes to the garage *)
                in
                    Sync; CabLoop[Sync, DaskC, TgetinC, TgetoutC, CgoD, Cgarage, Cout]
                where

                    (* Loop part of the timethread Cab *)
                  process CabLoop[Sync, DaskC, TgetinC, TgetoutC, CgoD, Cgarage, Cout] : noexit :=
                        DaskC;
                        TgetinC;
                        TgetoutC;
                        (
                            CgoD; CabLoop[Sync, DaskC, TgetinC, TgetoutC, CgoD, Cgarage, Cout]
                            []
                            Cgarage;
                            Cout; Cab2[Sync, DaskC, TgetinC, TgetoutC, Cout]
                        )
                    endproc (* CabLoop *)

            endproc (* Cab2 *)
    endproc (* Cab *)

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

(* Timethread_Plane *)
    process Plane[Pready, TgetonP, TgetoffP, Phangar] : noexit :=
        (* no hidden action in the timethread *)
        Pready;
        (
            TgetonP;
            TgetoffP;
            Phangar; stop
            |||
            (* recursive call *)
            Plane[Pready, TgetonP, TgetoffP, Phangar]
        )
    endproc (* Plane *)

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

(* Timethread Intermediate (Traveler and Cab) *)
    process TravCab[TgetinC, TgetoutC] : noexit :=
        (* hidden action *)
        hide
            TCride  (* Traveler takes a taxi ride *)
        in
```

```
        TgetinC;
        (
            TCride;
            TgetoutC; stop
            |||
            (* recursive call *)
            TravCab[TgetinC, TgetoutC]
        )
    endproc (* TravCab *)

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

(* Timethread Intermediate (Traveler and Plane) *)
    process TravPlane[TgetonP, TgetoffP] : noexit :=
        (* hidden action *)
        hide
            TPflight (* Flight of the traveler on the plane *)
        in
        TgetonP;
        (
            TPflight;
            TgetoffP; stop
            |||
            (* recursive call *)
            TravPlane[TgetonP, TgetoffP]
        )
    endproc (* TravPlane *)

endspec (* Traveler_Example *)
```

Appendix B: Simulation using Eludo and LOLA

I) Complete trace of a level 1 specification obtained with ELUDO:

```
---------------------------------- ACTIONS ----------------------------------
DEADLOCK
```

No more action possible

```
---------------------------------- HISTORY ----------------------------------
Traveler_Example[Tnew, Tdest]()
+-hidden Din;
  +-hidden Sync;
    +-hidden Cin;
      +-hidden Sync;
        +-Tnew;
          +-hidden Pready;
            +-hidden TphoneD;
              +-hidden DlookforC;
                +-hidden Dfillstat
                  +-hidden Dout;
                    +-hidden DaskC;
                      +-hidden TgetinC;
                        +-hidden TCride;
                          +-hidden TgetoutC;
                            +-hidden Cgarage;
                              +-hidden Cout;
                                +-hidden Tairport;
                                  +-hidden TgetonP;
                                    +-hidden TPflight;
                                      +-hidden TgetoffP;
                                        +-hidden Phangar;
                                          +-Tdest;
                                            +-DEADLOCK
ACTIONS    : <RET>execute, <v>iew, <h>istory, <s>ela, <g>oal, <q>uit, <o>ptions
```

Execution tree

The traveler has arrived to the destination.

No more action possible

II) Complete trace of a level 3 specification obtained with LOLA:

```
...
 [  1]  i; (* phangar *)

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> 1

  ==> i; (* phangar *)

------------------------------------------------------------------------

   DEADLOCK

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

 [  1] - i; (* din *)
 [  1] - i; (* sync *)
 [  1] - i; (* cin *)
 [  1] - i; (* sync *)
 [  1] - tnew;
 [  1] - i; (* tphoned *)
 [  1] - i; (* dlookforc *)
 [  1] - i; (* daskc *)
 [  1] - i; (* dfillstats *)
 [  1] - i; (* dready *)
 [  1] - i; (* tgetinc *)
 [  1] - i; (* tcride *)
 [  1] - i; (* tgetoutc *)
 [  1] - i; (* cgod *)
 [  1] - i; (* tairport *)
 [  1] - i; (* pready *)
 [  1] - i; (* tgetonp *)
 [  1] - i; (* tpflight *)
 [  1] - i; (* tgetoffp *)
 [  1] - tdest;
 [  1] - i; (* phangar *)

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

   Step-by-step simulation finished.

lola> q
quit
```

No more action possible

Execution tree

The traveler has arrived to the destination.

No more action possible

III) Partial trace of a level 3 specification obtained with LOLA:

```
[  1] - i; (* din *)
[  1] - i; (* sync *)
[  2] - i; (* cin *)
[  2] - i; (* sync *)
[  3] - tnew;
[  2] - i; (* tphoned *)
[  3] - i; (* cin *)
[  2] - i; (* dlookforc *)
[  2] - i; (* daskc *)
[  2] - i; (* dfillstats *)
[  3] - i; (* dout *)
[  3] - i; (* tgetinc *)
[  3] - i; (* tcride *)
[  3] - i; (* tgetoutc *)
[  7] - i; (* pready *)
[  4] - i; (* cgarage *)
[  3] - i; (* cout *)
[  4] - i; (* tairport *)
[  4] - i; (* tgetonp *)
[  6] - i; (* tpflight *)
[  4] - i; (* tgetoffp *)
[  6] - i; (* phangar *)
[  4] - tdest;
```

The traveler has arrived to the destination.

Other actions still possible since we can have multiple instances.

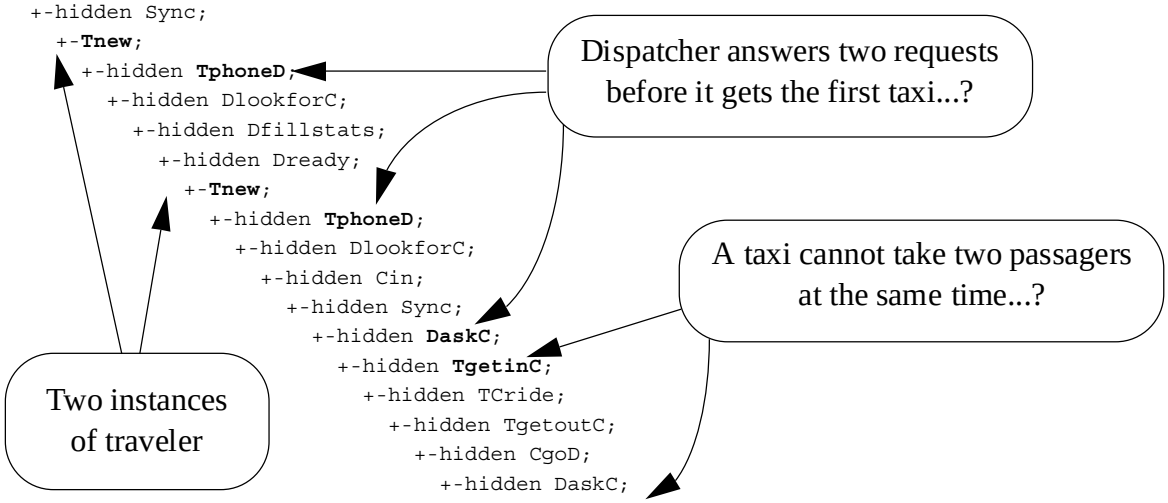
```
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> e

   Step-by-step simulation finished.

lola> quit
```

IV) Taxi ambiguities detected using ELUDO

```
---------------------------------- ACTIONS ----------------------------------
hidden Din; at line(s) 76
hidden Dfillstats; at line(s) 96
hidden Cin; at line(s) 116
hidden TCride; at line(s) 170
hidden Tairport; at line(s) 57
Tnew; at line(s) 50
---------------------------------- HISTORY ----------------------------------
Traveler_Example[Tnew, Tdest]()
+-hidden Din;
  +-hidden Sync;
    +-Tnew;
    +-hidden TphoneD;
      +-hidden DlookforC;
        +-hidden Dfillstats;
          +-hidden Dready;
            +-Tnew;
              +-hidden TphoneD;
                +-hidden DlookforC;
                  +-hidden Cin;
                    +-hidden Sync;
                      +-hidden DaskC;
                        +-hidden TgetinC;
                          +-hidden TCride;
                            +-hidden TgetoutC;
                              +-hidden CgoD;
                                +-hidden DaskC;
                                  +-hidden TgetinC;
ACTIONS    : <RET>execute, <v>iew, <h>istory, <s>ela, <g>oal, <q>uit, <o>ptions
```



V) Number of instances problem detected using ELUDO

```
---------------------------------- ACTIONS ----------------------------------
hidden Pready; at line(s) 149
hidden TphoneD; at line(s) 91,52
hidden TphoneD; at line(s) 91,52
hidden TphoneD; at line(s) 91,52
hidden TphoneD; at line(s) 91,52
hidden TphoneD; at line(s) 91,52
---------------------------------- HISTORY ----------------------------------
Traveler_Example[Tnew, Tdest]()
+-Tnew;
  +-Tnew;
    +-Tnew;
      +-Tnew;
        +-Tnew;
          +-hidden Din;
            +-hidden Sync;

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

Can the dispatcher really receive messages from the five travelers in any order he wants...?

Five instances of traveler are present. Does the system check for a bounded number of instances...?

VI) Interaction problem detected using ELUDO on a level 2 specification

```
---------------------------------- ACTIONS ----------------------------------
hidden Dfillstats; at line(s) 113
hidden TCride; at line(s) 188
hidden TgetoutC; at line(s) 152,189,73
hidden TgetoutC; at line(s) 152,189,73
hidden TgetoutC; at line(s) 152,189,73
hidden TgetoutC; at line(s) 152,189,73
---------------------------------- HISTORY ----------------------------------
Traveler_Mixed[Tnew, Tdest]()
+-Tnew;
  +-Tnew;
    +-hidden Pready;
      +-Tnew;
        +-hidden Cin;
          +-hidden Sync;
            +-hidden Cin;
              +-hidden Sync;
                +-hidden Din;
                  +-hidden Sync;
                    +-hidden TphoneD;
                      +-hidden DlookforC;
                        +-hidden DaskC;
                          +-hidden TgetinC;
                            +-hidden Dfillstats;
                              +-hidden Dready;
                                +-hidden TphoneD;
                                  +-hidden DlookforC;
                                    +-hidden DaskC;
                                      +-hidden TgetinC;
                                        +-hidden TCride;
ACTIONS    : <RET>execute, <v>iew, <h>istory, <s>ela, <g>oal, <q>uit, <o>ptions
```

These four similar choices indicate that the first traveler can get out the first cab *or* the second cab, and the samething applies to the second traveler. Is there an interaction problem...?

Two travelers got on two different cab. One of them had a ride.

VII) Symbolic extension of process *Traveler* in a transition system format

```
bh0 * 1 Tnew [50]
bh1 * | 1 TphoneD [52]
bh2 * | | 1 TgetinC [55]
bh3 * | | | 1 TgetoutC [56]
bh4 * | | | | 1 i (hiding: Tairport) [57]
bh5 * | | | | | 1 TgetonP [58]
bh6 * | | | | | | 1 TgetoffP [59]
bh7 * | | | | | | | 1 Tdest [60] DEADLOCK
    * | 2 TgetinC [55]
bh9 * | | 1 TphoneD [52] ==> again bh3
    * | | 2 TgetoutC [56]
bh10* | | | 1 TphoneD [52] ==> again bh4
    * | | | 2 i (hiding: Tairport) [57]
bh11* | | | | 1 TphoneD [52] ==> again bh5
    * | | | | 2 TgetonP [58]
bh12* | | | | | 1 TphoneD [52] ==> again bh6
    * | | | | | 2 TgetoffP [59]
bh13* | | | | | | 1 TphoneD [52] ==> again bh7
    * | | | | | | 2 Tdest [60]
bh14* | | | | | | | 1 TphoneD [52] DEADLOCK
```