Formal support for design techniques: a Timethreads-LOTOS approach

D. Amyot[*], *F. Bordeleau*[**], *R. J. A. Buhr*[**], *L. Logrippo*[*]

[*]*Telecommunication Software Engineering Research Group*
Department of computer science, University of Ottawa
Ottawa, Ont., Canada K1S 9B4
email: {damyot | luigi }@csi.uottawa.ca

[**]*Real-Time and Distributed Systems Research Group*
Department of systems and computer engineering, Carleton University
Ottawa, Ont., Canada K1S 5B6
email: {francis | buhr}@sce.carleton.ca

Abstract

A design methodology which allies the graphical expressiveness of the timethread notation with the analytical power of the LOTOS language and its associated tools is presented. The concept of timethread is at the basis of a design methodology based on scenarios. A simple telephone system is used as an example. It is shown how the main scenarios of such a system can be expressed by the timethread notation, leading to an abstract system design. Further, it is shown how the notation can be translated into LOTOS. LOTOS tools are used to validate the high-level design. Tools used include LOLA for analysis and design testing, LMC for checking temporal logic properties, and GOAL for checking reachability of actions.

Keywords

FDT-based software engineering, tools and tool support, design and design validation, timethreads, LOTOS

1 INTRODUCTION

1.1 Context and motivation

A software design methodology should meet two criteria: expressiveness and flexibility of the design language, and power of analysis and validation methods. For this reason, one seeks design methods that are based on expressive visual design notations, and formal analysis methods that are based on sound theoretical foundations. Design methods are intended to be used by system designers (architects or engineers) to describe systems (or system properties, such as scenarios, architecture and data transformation), while formal methods are used to verify that the system has the desired properties.

Timethreads (Buhr and Casselman, 1992 and 1995) are a high-level design notation for distributed systems that expresses scenario paths. In this paper, we show how the formal language LOTOS and its associated analysis methods and tools can be used to analyze and validate timethreads visual design descriptions.

Scenario-based approaches are now widely used in industry for the design of distributed systems. One of the main reasons is that scenarios describe top-level critical requirements that need to be fulfilled by any detailed design, and thereafter by implementations. Also, scenarios can usually be obtained easily from requirements. They express sequences of activities that need to be executed within the system in order to produce correct outputs from triggering events.

The concept of timethreads has been defined to be used in early stages of design (high-level design) to capture the different *scenario paths* that should drive the design process. They are used as a thinking tool in the requirement analysis phase where system designers try to understand the set of requirements as a whole before stepping into the detailed-design phase. Timethreads are defined as scenario paths because they illustrate paths along which scenarios flow in the system.

One of the particularities of the timethread methodology is that individual timethreads can be composed into a diagram called *timethread map*. Also, unlike other models for scenario description, timethreads make abstraction of specific mechanisms of component interaction. They provide for a notion of refinement of *activities* from a level of abstraction to the next.

Timethreads exist independently of any system structure, or decomposition. However, they are usually superimposed on structures, in which case they illustrate the sequences of activities through the set of system components. In this case, timethread activities, also called *responsibilities* in the timethread literature, are assigned to system components that become responsible to execute them in the detailed-design.

Space does not allow us to mention all aspects of the timethread methodology. Suffice is to say that it covers different stages of high-level design development.

The ISO standard FDT (Formal Description Technique) LOTOS (ISO, 1988) is used in this project for formal analysis and validation purpose. The reasons for choosing LOTOS are multiple. LOTOS allows to express both individual timethreads, as LOTOS processes, and interactions between timethreads in timethread maps, as LOTOS process interactions. It is executable with a formal operational semantics. Also, LOTOS possesses a *hide* operator that allows the designer to explicitly hide some gates in the specification, without having to modify the rest of the specification. This allows designers to focus on certain sets of activities when executing a specification. Finally, since LOTOS is an ISO standard, the number of tools that support it and the analysis and validation power of these tools are constantly increasing.

Research on integrating LOTOS in a design discipline which was a forerunner of timethreads was described in (Vigder and Buhr, 1992). Timethreads were then called *slices*. In the context of object-oriented systems (Buhr and Casselman, 1995), the authors renamed timethreads as *use case paths*.

1.2 Objectives

The objective of this project is to define a formal framework that will allow designers to analyze and validate a high-level design (a timethread map) against requirements (see Figure 1).

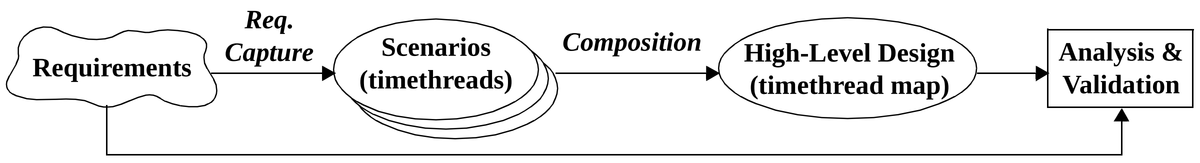


Figure 1 Validation of high-level design w.r.t. requirements.

This paper gives an overview of our formal framework for timethreads. Our approach to generate LOTOS specifications from timethread maps is based on the concept of *formal interpretation methods* defined in (Bordeleau, 1993). Formal interpretation methods allow the mapping of design models onto formal models that can be analyzed and validated. The LOTOS interpretation method for timethreads, which is described in detail in (Amyot, 1994), is briefly described in this paper (Section 2). We illustrate the use of the interpretation method by giving part of the LOTOS specification resulting from the interpretation of the timethread map of a case study: a simplified telephone system. We discuss how the resulting specification is obtained.

One important aspect of timethread maps that can be analyzed using our formal framework is the emergence of new unexpected scenarios, i.e. scenarios that were not intended in the timethread map, but that emerge from the interaction of timethreads that are described in the map. Such unexpected scenarios can be acceptable or not in the context of the system. The identification of such scenarios is critical in distributed system design, and it constitutes a difficult problem which is sometimes called *feature interaction problem*. We also analyze timethread maps with respect to deadlocks, non-deterministic sequences of activities and race conditions. We briefly discuss a few LOTOS tools used for the analysis and validation of timethread maps.

For simplicity, this paper emphasizes scenario aspects and makes abstraction from the notion of *components*, although components are shown in the timethreads maps, and although the timethread methodology has a well-defined role for them.

2 METHODOLOGY

2.1 LOTOS Interpretation Method for timethreads

In this section, we briefly describe the LOTOS interpretation method for timethreads that has been defined in (Bordeleau and Amyot, 1993). In the context of our interpretation method, we use the language TMDL (*Timethread Map Description Language*) for the textual description of timethread maps. The method, illustrated in Figure 2, allows the generation of LOTOS specifications from timethread maps. This method is mainly composed of four different sub-methods: the timethread map decomposition (*M1*), the structure interpretation (*M2*), the single timethread interpretation (*M3*) and the LOTOS specification composition (*M4*).

TMDL Description of Timethread Maps: *TMDL* (Amyot, 1994) is used to represent timethread maps in a textual form. It is the starting point of our LOTOS interpretation method. This language expresses both the topology of interacting timethreads and individual timethreads in a map. A prototype tool has been implemented to compile TMDL descriptions into LOTOS specifications. TMDL itself will not be discussed much further in this paper, but it will be used it in two short examples to explain the transition from the graphical timethread map to the textual LOTOS specification.

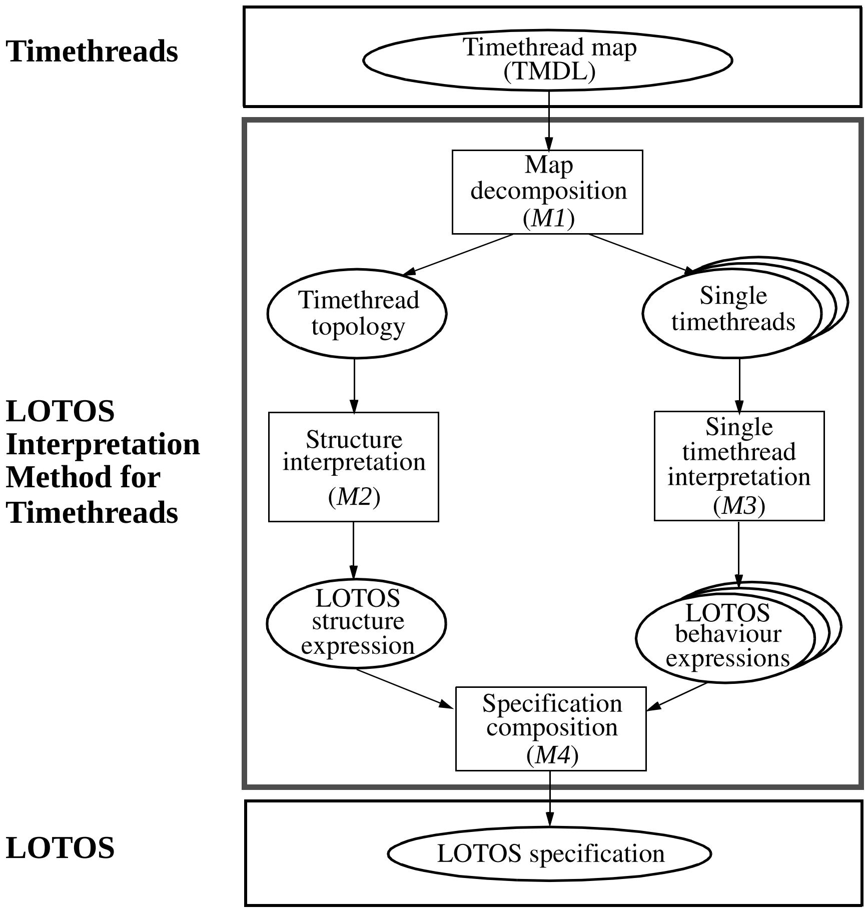

Timethreads
Timethread map (TMDL)
Map decomposition (*M1*)
Timethread topology
Single timethreads
LOTOS Interpretation Method for Timethreads
Structure interpretation (*M2*)
Single timethread interpretation (*M3*)
LOTOS structure expression
LOTOS behaviour expressions
Specification composition (*M4*)
LOTOS
LOTOS specification

Figure 2 LOTOS interpretation method for timethreads.

Timethread Map Decomposition Method (*M1*): Timethread maps are composed of a set of interacting timethreads. Formal interpretation methods deal separately with structure and behaviour. For this reason, the first step (*M1*) decomposes a timethread map into a set of individual timethreads (the `Descriptions` section of the TMDL description) and a timethread topology, i.e., a set of timethread interactions (the `Interactions` section of TMDL).

Structure Interpretation Method (*M2*): The goal of this method is to generate a LOTOS structure expression which expresses the topology of interacting timethreads in a system. The generation of this LOTOS expression is based on a method defined in (Bordeleau, 1993). Other similar methods, where graph structures are transformed into LOTOS expressions (trees), can be found in (Bolognesi, 1990, de Frutos-Eserig, 1993, and Hinterplattner and al., 1993).

Single Timethread Interpretation Method (*M3*): This method aims at generating LOTOS behaviour expressions from single timethreads. Each behaviour expression is a LOTOS process corresponding to its associated timethread.

Specification Composition Method (*M4*): We collect the structure expression and the set of behaviour expressions previously generated to produce a LOTOS specification.

2.2 Analysis and validation in LOTOS

Being formal and algebraic in nature, LOTOS lends itself to validation activities, which often are based on bisimulation concepts. This can help verifying that two specifications at two different levels of abstraction, or having different structures, are indeed comparable. If a LOTOS specification has a finite model, temporal logic analysis, such as model-checking, is possible. Being executable, a LOTOS specification produces a prototype of the entity specified, prototype which can be analyzed and tested (design-level testing). This opens a number of possibilities for validation, of which a few are demonstrated later in this paper.

In our methodology, a formal interpretation model (LOTOS) is obtained from a semi-formal notation (Timethreads). Therefore, the correctness of a timethread-to-LOTOS translation cannot be ensured.

3 EXAMPLE OF A SIMPLIFIED TELEPHONE SYSTEM

3.1 System description

We use the connection phase of a simplified telephone system to illustrate the methodology because most people intuitively know and understand this application. An *Initiator* tries to establish a connection with a *Responder* via a simple *Switch*, and several results, such as a completed connection, a disconnection, a wrong number, a busy tone, etc., can occur. We hereby define the activities that are used in our example as the requirements:

The Initiator:

- *OffHook*: picks the phone up. This is the first activity that initiates the connection phase.
- *HangUp*: hangs up. This activity can occur any time after a *OffHook*. It cancels the connection and disconnects the initiator.
- *Tone*: hears the dial tone. It occurs after *RegInit*.
- *Dial*: dials the phone number of the responder. At this level of abstraction, we assume it to be an atomic activity. It occurs after *Tone*.
- *ErrorDial*: receives an indication telling that an error occurred while dialing. The only option afterwards is for the initiator to hang up. It occurs if *Dial* does not occur in time.
- *WrongNumber*: receives an indication that the number does not exist. The only option afterwards is for the initiator to hang up. It occurs as a result of *CheckDB*.

- *BusyTone*: receives an indication telling that the responder is already using her phone. The only option afterwards is for the initiator to hang up. It occurs as a result of *CheckDB*.
- *NoAnswer*: receives an indication telling that the responder has not responded in time. The only option afterwards is for the initiator to hang up. It occurs if *Answer* does not occur in time

The Responder:

- *Ring*: can hear the phone ringing. It occurs after *CheckDB* if the responder is free.
- *Answer*: picks up the phone to answer the call. It can occur after *Ring*.

The Switch:

- *RegInit*: registers and initializes the user in its database. First activity after *OffHook*.
- *CheckDB*: checks the responder in the database. Indicates whether she exists or not, and whether her phone is already busy or not. It occurs after a correct dialing.
- *Connected*: indicates the expected end of the connection phase. Results from a correct answering.
- *Disconnected*: goes to this state after *HangUp* in connection phase, and updates the database.

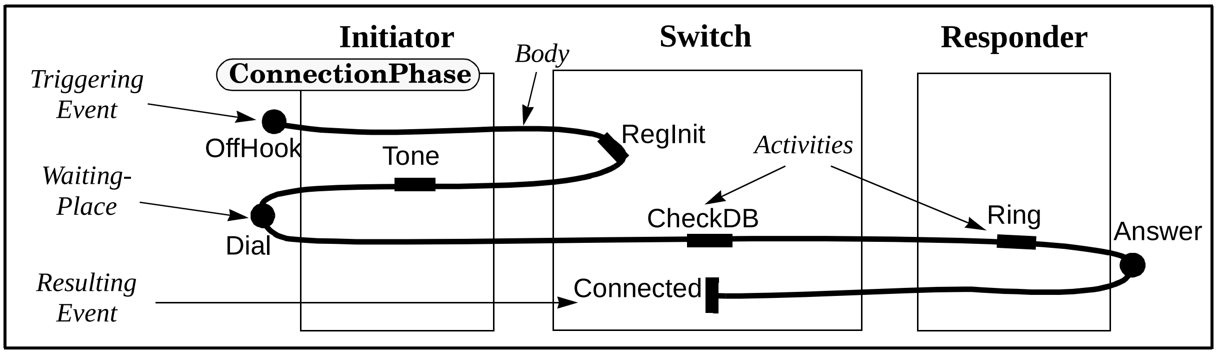
Note that our presentation is at an abstraction level where detailed-design mechanisms are not considered, for instance we do not consider the number of users. Such items can be considered later in the design process.

3.2 Timethread map

In this section, we introduce the Timethread methodology by describing how single timethreads and composed timethread maps are constructed from requirements.

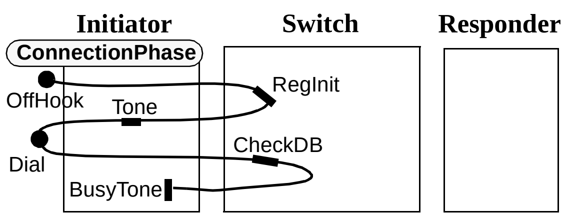
Single timethreads

The notation used to construct single timethreads includes several basic symbols that are identified in the Timethread map 1. Such a typical timethread starts with a *waiting-place* (triggering event ●) and ends with a *junction point* (resulting event ▮). The *body*, on which *activities* are sequentially placed, links the triggering event to its resulting event(s). Notions such as choice, parallelism, time-out and synchronization can also be expressed.

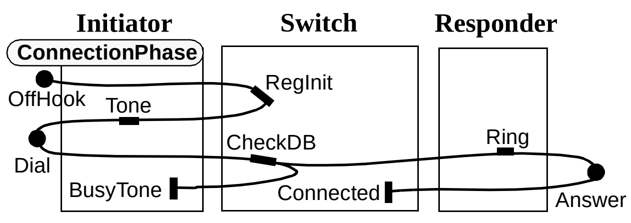


Timethread map 1: Successful connection.

This first timethread presents the traditional scenario path of a successful connection phase. The second timethread (Timethread map 2) illustrates the path where the initiator receives a busy signal, as a result of the responder's phone being already busy. Other timethreads could be derived from the requirements, but we will consider these two only to illustrate the key features of the methodology.



Timethread map 2: Initiator gets a busy tone.



Timethread map 3: Merging of Map 1 and Map 2.

Composed timethread maps

The methodology allows the composition of multiple paths in a single map. The result of timethread composition is either the merging of several timethreads into a single one that offers alternate paths (choice), e.g. Timethread map 3, or the connection of timethreads in a single map, e.g. Timethread map 4. In the latter case, timethreads interact using internal events. Merged and connected timethreads provide new scenario paths resulting from the composition of simpler ones.

Timethread map 3 presents a merging of maps 1 and 2; they have a common sequence of activities and then a choice has to be made based on the system state (information received from *CheckDB*) in order to follow one path or the other.

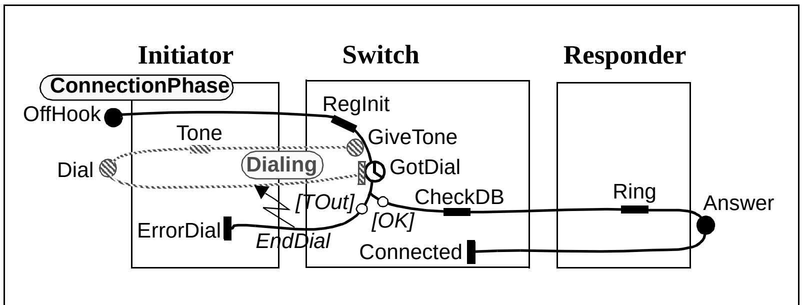
Interactions may occur on waiting-places and junction points. For instance, two synchronous interactions occur between timethreads *ConnectionPhase* and *Dialing* on the waiting-places *GiveTone* and *GotDial* (two new internal activities) in Timethread map 4.

The *in-passing* semantics of interactions is visually represented by the triggering event of a timethread *TT1* being juxtaposed to the body of a timethread *TT2*. When *TT2* reaches this point on the path, it triggers *TT1* and then continues. This is the case for activity *GiveTone* in Map 4. A waiting-place on a body waits for an event to occur from the environment (ex.: *Dial*) or from the result of another timethread (ex.: *GotDial*).

Several types of transformations, such as splitting, extension, and refinement, can be applied on timethread maps. Map 4 is the result of two such transformations applied to Map 1.

The successful connection has been **extended** with a new alternative result (*ErrorDial*), caused by a time-out during the dialing. This time-out occurs at the time waiting-place (◷) *GotDial*, which then returns *TOut* (a *tag*, or a value) as a result. Note that guards (*[TOut]* and *[OK]*) have been added on the alternative paths. Then, a segment of the path is **split** to allow parallelism between the dialing sequence and the rest of the path starting with the time waiting-place. The timethread *Dialing* is therefore created and finally composed with the original *ConnectionPhase* timethread on two synchronization points.

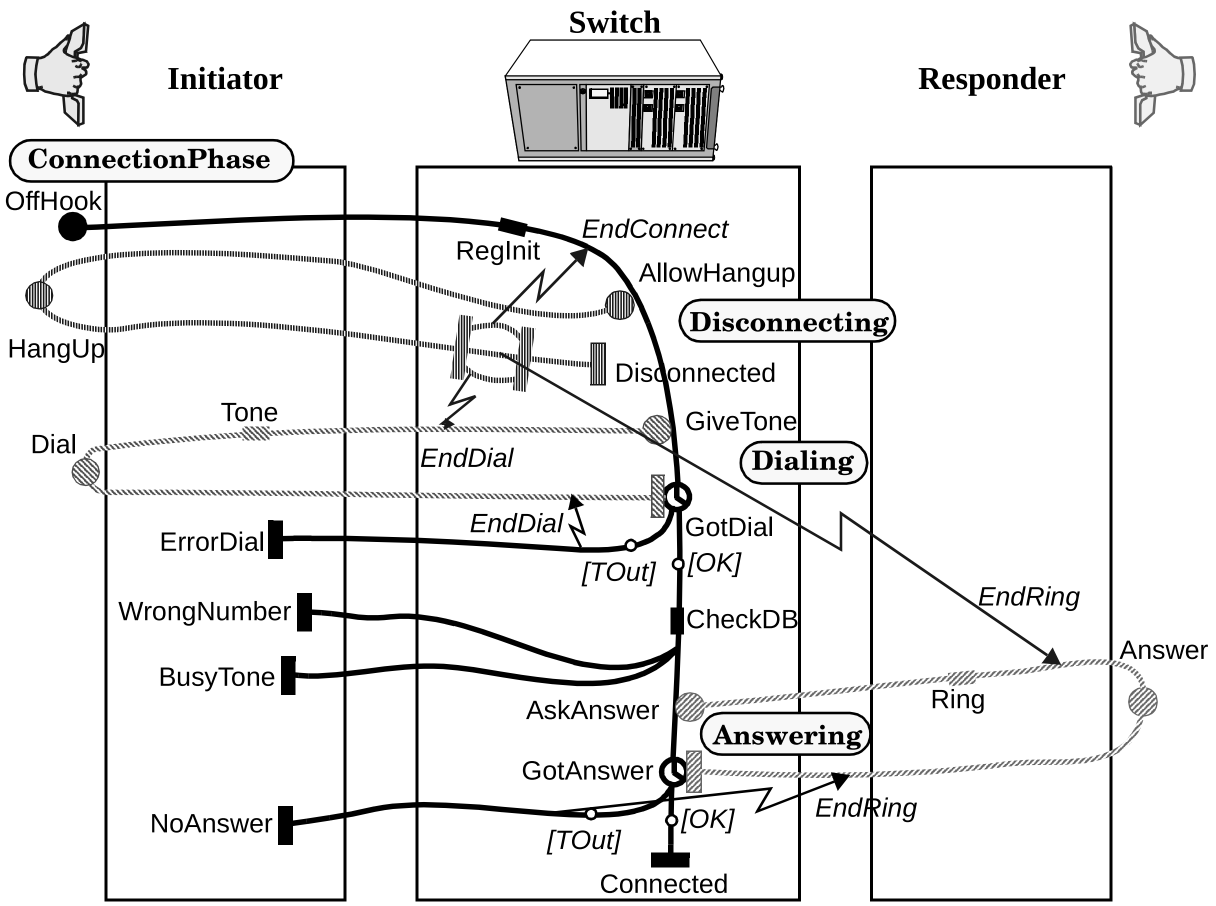
The broken arrow (↘) represents an abort activity, which means that *EndDial* (an internal activity) will destroy or disable ongoing scenarios along the timethread *Dialing*, similarly to a reset feature.



Timethread map 4: Splitting and extension of Map 1.

Final timethread map

Along the way, we could create many single timethreads and compose or transform them into more complex timethread maps, as we did for Timethread map 4. We do so until we reach the point where we have captured all the requirements. The final Timethread map 5 represents the requirements at the level of abstraction chosen for this paper. Although space constraints forbid us to cover here the complete process that led to this map, we raise several points to give the reader a better understanding of the timethread methodology and the operations involved.



Timethread map 5: Simple telephone connection (Final map).

Our design process resulted in four interconnected timethreads:

- *ConnectionPhase*: Gives the initiator the result of the triggering event *OffHook*. Triggers the three other timethreads along the path.
- *Disconnecting*: Manages the *HangUp* activity and then returns to a disconnected state.
- *Dialing*: Manages the initiator's activity *Dial*.
- *Answering*: Manages the responder's activity *Answer*.

To compose these timethreads (synchronously), several new internal events were created:

- *AllowHangup*: Triggers the *Disconnecting* timethread that allows the *HangUp* event to be received.
- *GiveTone*: Triggers the *Dialing* timethread.
- *GotDial*: Indicates that the *Dial* arrived in time. This is a time waiting-place, so a time-out could occur (with the tag value *TOut* as a result).
- *AskAnswer*: Triggers the *Answering* timethread.
- *GotAnswer*: Indicates that the *Answer* arrived in time. This is a time waiting-place, so a time-out could occur (with the tag value *TOut* as a result).
- *EndConnect*: Aborts the timethread *ConnectionPhase*.
- *EndDial*: Aborts the timethread *Dialing*.
- *EndRing*: Aborts the timethread *Answering*.

In the *Disconnecting* timethread, we also find an instance of three activities (*EndConnect*, *EndDial*, *EndRing*) that are performed in parallel.

3.3 Application of the Interpretation Method

Once we have a satisfactory timethread map, we use the LOTOS interpretation method to generate the specification to be analyzed and validated against the system requirements.

As a result of the timethread map decomposition, we obtain a timethread topology such as the one in Figure 3(a), and a set of single timethreads descriptions, such as the one in Figure 4(a). Both come from their respective sections in the TMDL description. We do not show the final map as an example because it involves too many details. Its TMDL description and LOTOS specification are available upon request to the authors.

Note that we can define internal activities both globally in the map and locally in a timethread (see the two examples). For that purpose, we use the LOTOS *hide* operator in the specifications. Any activity can be defined as internal, depending on the level of abstraction.

The second step in the LOTOS interpretation method consists in the application of the structure interpretation method, where a LOTOS structure is generated from the topology of interacting timethreads (the `Interactions` section in TMDL). In Figure 3(b), the result obtained from our compiler is shown. We can observe that timethread interactions are mapped onto synchronization gates in the LOTOS specification, and individual timethreads are mapped onto LOTOS processes.

(a) Map 4 structure: TMDL description

```
Map Map4 is

INTERNAL  { Hidden global activities }
   GiveTone, GotDial, EndDial

INTERACTIONS
   ConnectionPhase, Dialing
      on  GiveTone, GotDial, EndDial;

DESCRIPTIONS
   Timethread ConnectionPhase is
   { Single timethread description... }
   EndTT { ConnectionPhase }

   Timethread Dialing is
   { Single timethread description... }
   EndTT { Dialing }

EndMap { Scenario4 }
```

(b) Map 4 structure: LOTOS specification

```
specification Map4[Answer, Connected, Dial,
                   ErrorDial, OffHook, Ring, Tone]
                   :noexit

library
   Boolean, NaturalNumber
endlib

(* Tag ADT definition *)
type Tag is Boolean, NaturalNumber
   (* Type description... *)
endtype

behaviour

hide EndDial, GiveTone, GotDial in
   (
      ConnectionPhase[Answer, Connected, EndDial,
                      ErrorDial, GiveTone, GotDial,
                      OffHook, Ring]
      |[EndDial, GiveTone, GotDial]|
      Dialing[Dial, EndDial,GiveTone, GotDial,Tone]
   )

where
   process ConnectionPhase[...]:noexit :=
      (* Description... *)
   endproc  (* Timethread ConnectionPhase *)

   process Dialing[...]:noexit :=
      (* Description... *)
   endproc  (* Timethread Dialing *)

endspec (* Map Scenario4 *)
```

Figure 3   Map structure (Map 4) in TMDL and LOTOS.

In the third step (see Figure 4), we apply the single timethread interpretation method to generate LOTOS processes, or behaviour expressions, from their corresponding timethreads (from the `Interactions` section in the TMDL description). Note that triggers, waiting-places, activities (actions) and results all correspond to LOTOS gates. The identifiers used in the LOTOS specification are the ones used in the TMDL description. More complex TMDL operators make use of other LOTOS constructs such as choices, interleavings, synchronizations, disablings, ADTs and guards, recursion, etc. In the fourth step, we combine the LOTOS structure expression and behaviour expressions in a global LOTOS specification. The latter reflects the scenario paths induced by the timethread map, and is thereafter used by the analysis and validation tools.

(a) *ConnectionPhase*: TMDL timethread

```
Timethread ConnectionPhase is
   { This is the main timethread. }
   Internal
        RegInit,
        CheckDB

   Trigger (OffHook)
   Action (RegInit)
   Action (Tone)
   Wait (Dial)
   Action (CheckDB)
   Action (Ring)
   Wait (Answer)
   Result (Connected)
EndTT { ConnectionPhase }
```

(b) *ConnectionPhase*: LOTOS process

```
process ConnectionPhase[BusyTone, Dial, OffHook,
                        Tone]:noexit :=
hide CheckDB, RegInit in

   OffHook;
   (
      RegInit;
      Tone;
      Dial;
      CheckDB;
      Ring;
      Answer;
      Connected; stop  (* No recursion *)
   )
endproc  (* Timethread ConnectionPhase *)
```

Figure 4 Timethread *ConnectionPhase* (Map 1) in TMDL and LOTOS.

3.4 Formal analysis and validation

We now illustrate a number of LOTOS-based techniques and tools to check the high-level design. With available LOTOS tools, we can analyze and test whether the specification accepts all the scenarios from the requirements and refuses undesirable scenarios. We use model-checking and goal-oriented execution to verify properties such as absence of deadlocks and race conditions. We present in this section several key examples related to our case study.

Analysis

The composition of timethreads results in the emergence of new scenarios, sometimes undesirable. The analysis of the specification corresponding to the final composed map helps in discovering these scenarios. LOTOS analysis can be done using step-by-step execution of the specification with tools such as ELUDO (Ghribi and Logrippo, 1993) and LOLA (Quemada, Pavón, and Fernández, 1988). We used LOLA in the following two examples.

Trace 1 in Figure 5(a) shows a global and expected scenario leading to a successful connection. Internal actions starting with a `Sync_` are automatically added by our TMDL-to-LOTOS compiler for internal synchronizations. For instance, `Sync_time_1` checks whether there is a time-out or not.

Trace 2 presents a potential problem discovered in the high-level design. We see that a *Dial* can be followed by an *ErrorDial*, although this is forbidden in the requirements. The error is caused by a race condition at the waiting-place *GotDial*. A time-out occurred right after *Dial* was performed and before *GotDial* was received. Other such problems can be found during the analysis phase.

The problems and issues discovered during the analysis (and also during testing and verification) do not have to be solved at a timethread level. This might be too high a level of abstraction to do so. The goal here is to discover potential causes of problems that must be solved during the detailed-design phase.

(a) Trace 1: Successful connection	(b) Trace 2: Race condition while dialing
`[  1] - offhook;`	`[  1] - offhook;`
`[  1] - i; (* reginit *)`	`[  1] - i; (* reginit *)`
`[  1] - i; (* allowhangup *)`	`[  1] - i; (* allowhangup *)`
`[  1] - i; (* givetone *)`	`[  1] - i; (* givetone *)`
`[  3] - tone;`	`[  3] - tone;`
`[  3] - dial;`	`[  3] - dial;`
`[  2] - i; (* gotdial *)`	`[  1] - i; (* timeout_0 *)`
`[  1] - i; (* sync_time_1 ! ok *)`	`[  1] - i; (* sync_time_1 ! tout *)`
`[  1] - i; (* checkdb *)`	`[  1] - i; (* enddial *)`
`[  3] - i; (* askanswer *)`	`[  1] - errordial;`
`[  3] - ring;`	
`[  3] - answer;`	
`[  2] - i; (* gotanswer *)`	
`[  1] - i; (* sync_time_3 ! ok ! ok *)`	
`[  1] - connected;`	

Figure 5 Traces resulting from step-by-step execution with LOLA.

Testing

Step-by-step execution is useful for an intuitive discovery of problems, but the global state space is usually too large for this technique to be efficient. One way to reduce this state space is to compose the specification with a test case, hence allowing an exhaustive search for problems in this context.

Design testing allows us to test internal activities of components by making them visible, in a grey-box fashion. One can check whether or not test cases that must be accepted can be executed by the specification, and whether or not test cases that must be rejected are always forbidden by the specification. This is called acceptance/rejection testing. Our methodology allows us to decide which activities should be internal and which should not, therefore leading to a high level of flexibility for the definition of test cases.

Single timethreads defined while constructing the final map can be used as acceptance test cases. These usually are straightforward scenarios that must be found in the composed map. Figure 6(a) presents a successful-connection test case derived, using the single timethread interpretation method, from the timethread in Map 1. We used the testing capabilities of LOLA to validate *Test1* against the specification, and the result was a *MAY PASS* verdict. We can see from Figure 6(b) that 355 scenarios (traces) are possible, from which 20 lead to successes and 335 lead to deadlocks. This result is due to the combinations of possibilities where a time-out occurs (at *GotDial* or *GotAnswer*). Figure 6(c) presents one of these unexpected scenarios that were obtained from LOLA.

(a) Acceptance Test 1: LOTOS process

```
process Test1[Answer,Connected,Dial,
              OffHook, Ring, Tone,
              Success]:noexit :=
(* Comes directly from map 1 *)
   hide CheckDB, RegInit in
      OffHook;
      (
         RegInit;
         Tone;
         Dial;
         CheckDB;
         Ring;
         Answer;
         Connected;
         Success; stop
            (* No recursion *)
      )
   endproc  (* Test1 *)
```

(b) Execution on LOLA

```
lola> TestExpand 100 Success
      Test1 -y -i

  Analysed states       = 1083
  Generated transitions = 1102
  Duplicated states     = 0
  Deadlocks             = 335

  Process Test = test1
  Test result  = MAY PASS.

  355 executions analysed:

              successes = 20
                  stops = 335
                  exits = 0
          cuts by depth = 0
```

(c) Example of Unexpected scenario

```
offhook;
i; (* reginit *)
i; (* reginit *);
i; (* allowhangup *)
i; (* givetone *)
tone;
dial;
i; (* checkdb *)
i; (* timeout_0 *)
i; (* sync_time_1
      ! tout *)
i; (* enddial *)
stop
```

Figure 6 Acceptance Test 1, derived from Map 1.

Test cases can also be generated by hand in LOTOS. For instance, *Test2* (Figure 7) is an acceptance test case checking that connections are impossible when the responder does not perform the *Answer* activity. The result shows that 12 scenarios successfully satisfy the test and that there is no rejection. Therefore the verdict is a *MUST PASS*, as expected.

Rejection test cases are LOTOS processes that must deadlock in all cases when composed with the specification. Such tests could also be defined for our case study.

(a) Acceptance Test 2: LOTOS process

```
process Test2[BusyTone, Dial, OffHook, Tone,
             Success,NoAnswer,ErrorDial]:noexit:=
(* More complex test checking that we cannot be *)
(* connected if the responder does not answer *)
      OffHook;
      (
         (
            Tone;
            Dial;
            BusyTone; Success; stop
                                  (* No recursion *)
         )
         [> (ErrorDial; Success; stop
             []
             NoAnswer; Success; stop)
      )
   endproc  (* Test2 *)
```

(b) Execution on LOLA

```
lola> TestExpand 100 Success Test2 -y -i
    Analysed states       = 52
    Generated transitions = 63
    Duplicated states     = 0
    Deadlocks             = 0

    Process Test = test2
    Test result  = MUST PASS.

               successes = 12
                   stops = 0
                   exits = 0
           cuts by depth = 0
```

Figure 7 Acceptance Test 2.

Verification using model-checking

Model-checking (Clarke, Emerson, and Sistla, 1986) is a technique for verifying the truth or falsehood of temporal logic properties in a model of the specification. With this technique, we can skip non-essential activities, and thus concentrate on the important ones, in the expression of a property. If test cases are use, instead, all intermediate external activities between two activities of interest must be included.

Some of the temporal logic quantifiers allowed are `AG` (henceforth in all futures), `AF` (eventually in all futures), `EF` (eventually in some future). LMC (the LOTOS Model Checker (Ghribi, 1992)) is a model-checking tool that is part of the University of Ottawa LOTOS toolkit ELUDO. A finite model of the specification is obtained by the tool SELA (also part of ELUDO) and then transformed into a Kripke model by LMC. Some of the design properties checked by using LMC were:

- `EF ( 'Connected' )`: A connection will occur eventually in some future. This is a requirement that should be satisfied by any design. LMC indicates that this formula holds.
- `AG ('Dial'` $\rightarrow$ `AF ('WrongNumber'`$\vee$`'BusyTone'`$\vee$`'NoAnswer'`$\vee$`'Connected'))`: All *Dial* will be followed (not necessarily immediately) by one of the four results enumerated. LMC returns a **false** verdict, therefore indicating a problem. There is a race condition at the time waiting-place *GotDial* and the resulting event could be *ErrorDial*. This problem was previously found in the analysis, as illustrated by the second trace of Figure 5.
- `AG ( 'Answer'` $\rightarrow$ `AF ('Connected') )`: Every *Answer* will be followed (not necessarily immediately) by a *Connected*. Although this property sounds intuitively correct, LMC's verdict is **false** because of a race condition at the time waiting-place *GotAnswer*, thus avoiding the *Connected* result.
- `EF ( 'HangUp'` $\rightarrow$ `EF ('Connected') )`: A *HangUp* may be followed by a *Connected*. This is an example of a negative property that must be rejected by our design. LMC indicates that this formula holds, so we conclude that there is another problem at this level. In this case, a *Connected* could slip between *HangUp* and *EndConnect*, indicating a third race condition.

Many more positive and negative properties derived from the requirements can be verified with model-checking, and other problems than race conditions can be found (non-determinism, wrong ordering, deadlocks, etc.).

Goal-oriented execution

One disadvantage of model-checking is that it requires a complete model of the system to be constructed. For realistic systems, such models are (at best) computationally expensive to obtain. Goal-oriented execution (Haj-Hussein, Sincennes and Logrippo, 1993, and Brinksma and Eertink, 1993) is a LOTOS execution mode that directs execution to see whether certain action sequences can be achieved. Obviously impossible search directions are avoided by our GOAL tool (also part of ELUDO). Here are several design requirements, similar to the properties previously presented using temporal logic, which GOAL allows to verify:

- `GOAL [Connected]`: This goal checks that a *Connected* result can be reached. This gives the set of possible scenarios ending with this activity.

- `GOAL [Dial, ErrorDial]`: This goal checks that a *Dial* can be reached, and that this can be followed by an *ErrorDial*. The requirements state that this situation should not happen. However GOAL indicates that it can happen, because of the race condition at *GotDial*. This goal is complementary to the second temporal logic property to obtain all the traces that lead to *Dial* and then to *ErrorDial*.
- `GOAL [Dial, Answer, NoAnswer]`: This goal checks that a *Dial* can be reached, after this an *Answer* can be reached, and eventually a *NoAnswer* can be reached. This goal outputs the traces that invalidate the third property expressed in the previous section.

Goals are useful by themselves to verify different properties, but they are also powerful as diagnostic tools for problems detected using model-checking.

4 CONCLUSION AND FUTURE WORK

We have demonstrated a design methodology which is based on the use of two complementary techniques: timethreads and process algebras (LOTOS). Timethreads are obtained from scenarios, which in turn are obtained from requirements, and then are combined into high-level designs. Timethread's graphical notation is translated manually into the language TMDL, and this automatically into LOTOS. LOTOS specifications can be analyzed, by using tools, for conformance to requirements. The essential points of this process were demonstrated on a simple telephony example. The validation techniques used were acceptance/rejection testing, model-checking, and goal-oriented execution. In (Amyot 1994), a much larger telepresence example was developed in a similar fashion.

Several research directions need to be pursued. Research must continue towards determining what are the validation techniques that are useful in relation with timethread design, and towards efficiently implementing them. Tools for the methodology must be implemented. In the long range, we see an integrated environment where this type of design and validation techniques can be carried out by users familiar with timethreads only.

5 ACKNOWLEDGMENT

This work was funded in part by the Telecommunications Research Institute of Ontario and by the Natural Science and Engineering Research Council of Canada. We are grateful to Michel Locas, Ron Casselman, Abdellatif Obaid, and Trevor Pearce for contributions that led to this article.

6 REFERENCES

Amyot, D. (1994) *Formalization of Timethreads Using LOTOS*. M.Sc. Thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.

Bolognesi, T. (1990) “A Graphical Composition Theorem for Networks of LOTOS Processes”. In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, 88-95.

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

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

Bordeleau, F. and Locas, M. (1994) "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.

Brinksma, E. and Eertink, H. (1993) "Goal-Driven LOTOS Execution". In: A. Danthine, G. Leduc, and P. Wolper (Eds), *Protocol Specification, Testing and Verification, XIII,* North-Holland, 45-60.

Buhr, R.J.A. and Casselman, R.S. (1995) *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA. To appear in October.

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

Clarke, E.M., Emerson, E.A. and A.P. Sistla (1986) "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications". *ACM TOPLAS*, 8(2), 244-263.

de Frutos-Eserig, D. (1993) "A Characterization of LOTOS Representable Networks of Parallel Processes". In: P. Scollo (Ed), *Proceedings of AMAST'93.*

Ghribi, B. (1992) *A Model Checker For LOTOS*. M.Sc. Thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.

Ghribi, B. and Logrippo, L. (1993) "A Validation Environment for LOTOS". In: A. Danthine, G. Leduc, and P. Wolper (Eds), *Protocol Specification, Testing and Verification, XIII,* North-Holland.

Haj-Hussein, M., Logrippo, L. and Sincennes, J. (1993) "Goal Oriented Execution for LOTOS". In: M. Diaz and R. Groz (Eds), *Formal Description Techniques, V,* North-Holland, 311-327.

Hinterplattner, J., Nirshl, H. and Saria H. (1991), "Process Topology Diagram", In: J. Quemada, J. Mañas, and E. Vázquez (Eds), *Formal Description Techniques, III*, North-Holland.

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

Quemada, J., Pavón, S. and Fernández, A. (1988) "Transforming LOTOS Specifications with LOLA: The Parametrized Expansion". In: K. J. Turner (Ed), *Formal Description Techniques, I*, IFIP/North-Holland, 45-54.

Vigder, M. and Buhr, R.J.A. (1992) "Using LOTOS in a Design Environment". In: K.R.Parker, G.A. Rose, (Eds), *Formal Description Techniques, IV,* IFIP/North-Holland, 1-14.

Most theses and technical reports referenced in this paper are available on the World Wide Web at *http://www.csi.uottawa.ca:80/~lotos/ and http://www.sce.carleton.ca/rads/doors.html.*

Formal support for design techniques: a Timethreads-LOTOS approach

D. Amyot[*], *F. Bordeleau*[**], *R. J. A. Buhr*[**], *L. Logrippo*[*]

[*]*Telecommunication Software Engineering Research Group*
Department of computer science, University of Ottawa
Ottawa, Ont., Canada K1S 9B4
email: {damyot | luigi }@csi.uottawa.ca

[**]*Real-Time and Distributed Systems Research Group*
Department of systems and computer engineering, Carleton University
Ottawa, Ont., Canada K1S 5B6
email: {francis | buhr}@sce.carleton.ca

Abstract

A design methodology which allies the graphical expressiveness of the timethread notation with the analytical power of the LOTOS language and its associated tools is presented. The concept of timethread is at the basis of a design methodology based on scenarios. A simple telephone system is used as an example. It is shown how the main scenarios of such a system can be expressed by the timethread notation, leading to an abstract system design. Further, it is shown how the notation can be translated into LOTOS. LOTOS tools are used to validate the high-level design. Tools used include LOLA for analysis and design testing, LMC for checking temporal logic properties, and GOAL for checking reachability of actions.

Keywords

FDT-based software engineering, tools and tool support, design and design validation, timethreads, LOTOS

1 INTRODUCTION

1.1 Context and motivation

A software design methodology should meet two criteria: expressiveness and flexibility of the design language, and power of analysis and validation methods. For this reason, one seeks design methods that are based on expressive visual design notations, and formal analysis methods that are based on sound theoretical foundations. Design methods are intended to be used by system designers (architects or engineers) to describe systems (or system properties, such as scenarios, architecture and data transformation), while formal methods are used to verify that the system has the desired properties.

Timethreads (Buhr and Casselman, 1992 and 1995) are a high-level design notation for distributed systems that expresses scenario paths. In this paper, we show how the formal language LOTOS and its associated analysis methods and tools can be used to analyze and validate timethreads visual design descriptions.

Scenario-based approaches are now widely used in industry for the design of distributed systems. One of the main reasons is that scenarios describe top-level critical requirements that need to be fulfilled by any detailed design, and thereafter by implementations. Also, scenarios can usually be obtained easily from requirements. They express sequences of activities that need to be executed within the system in order to produce correct outputs from triggering events.

The concept of timethreads has been defined to be used in early stages of design (high-level design) to capture the different *scenario paths* that should drive the design process. They are used as a thinking tool in the requirement analysis phase where system designers try to understand the set of requirements as a whole before stepping into the detailed-design phase. Timethreads are defined as scenario paths because they illustrate paths along which scenarios flow in the system.

One of the particularities of the timethread methodology is that individual timethreads can be composed into a diagram called *timethread map*. Also, unlike other models for scenario description, timethreads make abstraction of specific mechanisms of component interaction. They provide for a notion of refinement of *activities* from a level of abstraction to the next.

Timethreads exist independently of any system structure, or decomposition. However, they are usually superimposed on structures, in which case they illustrate the sequences of activities through the set of system components. In this case, timethread activities, also called *responsibilities* in the timethread literature, are assigned to system components that become responsible to execute them in the detailed-design.

Space does not allow us to mention all aspects of the timethread methodology. Suffice is to say that it covers different stages of high-level design development.

The ISO standard FDT (Formal Description Technique) LOTOS (ISO, 1988) is used in this project for formal analysis and validation purpose. The reasons for choosing LOTOS are multiple. LOTOS allows to express both individual timethreads, as LOTOS processes, and interactions between timethreads in timethread maps, as LOTOS process interactions. It is executable with a formal operational semantics. Also, LOTOS possesses a *hide* operator that allows the designer to explicitly hide some gates in the specification, without having to modify the rest of the specification. This allows designers to focus on certain sets of activities when executing a specification. Finally, since LOTOS is an ISO standard, the number of tools that support it and the analysis and validation power of these tools are constantly increasing.

Research on integrating LOTOS in a design discipline which was a forerunner of timethreads was described in (Vigder and Buhr, 1992). Timethreads were then called *slices*. In the context of object-oriented systems (Buhr and Casselman, 1995), the authors renamed timethreads as *use case paths*.

1.2 Objectives

The objective of this project is to define a formal framework that will allow designers to analyze and validate a high-level design (a timethread map) against requirements (see Figure 1).

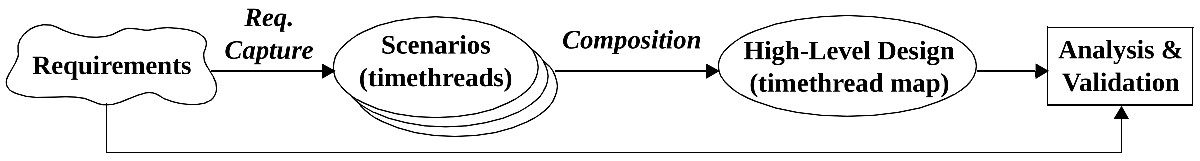


Figure 1 Validation of high-level design w.r.t. requirements.

This paper gives an overview of our formal framework for timethreads. Our approach to generate LOTOS specifications from timethread maps is based on the concept of *formal interpretation methods* defined in (Bordeleau, 1993). Formal interpretation methods allow the mapping of design models onto formal models that can be analyzed and validated. The LOTOS interpretation method for timethreads, which is described in detail in (Amyot, 1994), is briefly described in this paper (Section 2). We illustrate the use of the interpretation method by giving part of the LOTOS specification resulting from the interpretation of the timethread map of a case study: a simplified telephone system. We discuss how the resulting specification is obtained.

One important aspect of timethread maps that can be analyzed using our formal framework is the emergence of new unexpected scenarios, i.e. scenarios that were not intended in the timethread map, but that emerge from the interaction of timethreads that are described in the map. Such unexpected scenarios can be acceptable or not in the context of the system. The identification of such scenarios is critical in distributed system design, and it constitutes a difficult problem which is sometimes called *feature interaction problem*. We also analyze timethread maps with respect to deadlocks, non-deterministic sequences of activities and race conditions. We briefly discuss a few LOTOS tools used for the analysis and validation of timethread maps.

For simplicity, this paper emphasizes scenario aspects and makes abstraction from the notion of *components*, although components are shown in the timethreads maps, and although the timethread methodology has a well-defined role for them.

2 METHODOLOGY

2.1 LOTOS Interpretation Method for timethreads

In this section, we briefly describe the LOTOS interpretation method for timethreads that has been defined in (Bordeleau and Amyot, 1993). In the context of our interpretation method, we use the language TMDL (*Timethread Map Description Language*) for the textual description of timethread maps. The method, illustrated in Figure 2, allows the generation of LOTOS specifications from timethread maps. This method is mainly composed of four different sub-methods: the timethread map decomposition (*M1*), the structure interpretation (*M2*), the single timethread interpretation (*M3*) and the LOTOS specification composition (*M4*).

TMDL Description of Timethread Maps: *TMDL* (Amyot, 1994) is used to represent timethread maps in a textual form. It is the starting point of our LOTOS interpretation method. This language expresses both the topology of interacting timethreads and individual timethreads in a map. A prototype tool has been implemented to compile TMDL descriptions into LOTOS specifications. TMDL itself will not be discussed much further in this paper, but it will be used it in two short examples to explain the transition from the graphical timethread map to the textual LOTOS specification.

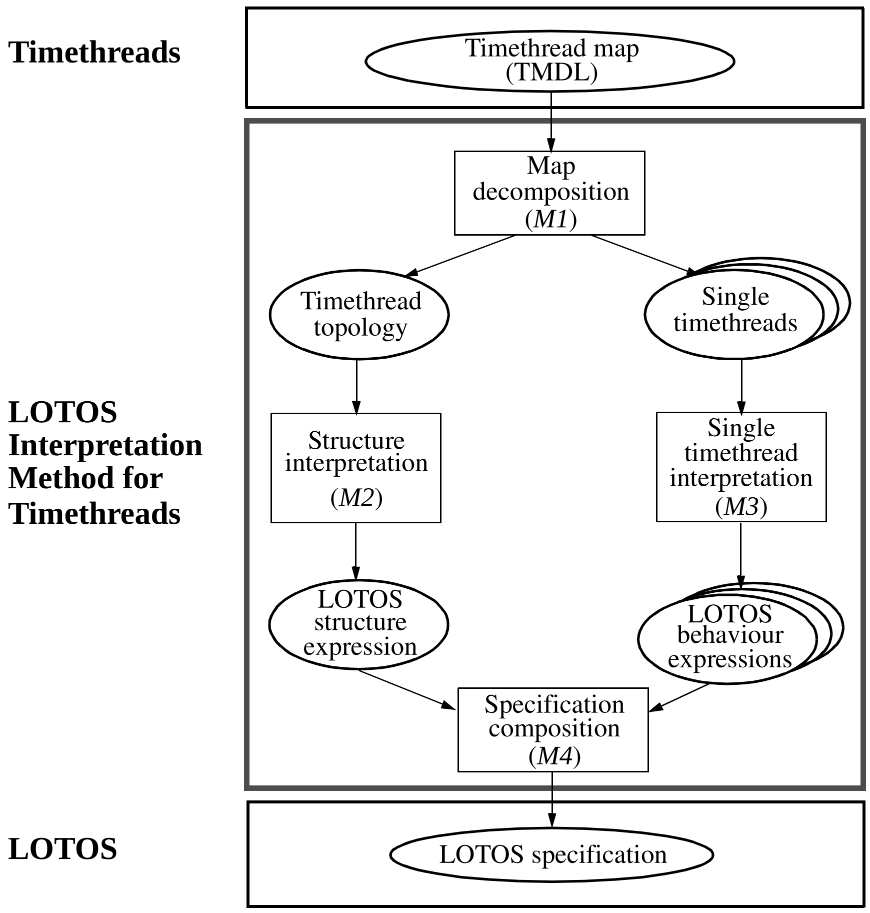

Timethreads
Timethread map (TMDL)
Map decomposition (*M1*)
Timethread topology
Single timethreads
LOTOS Interpretation Method for Timethreads
Structure interpretation (*M2*)
Single timethread interpretation (*M3*)
LOTOS structure expression
LOTOS behaviour expressions
Specification composition (*M4*)
LOTOS
LOTOS specification

Figure 2 LOTOS interpretation method for timethreads.

Timethread Map Decomposition Method (*M1*): Timethread maps are composed of a set of interacting timethreads. Formal interpretation methods deal separately with structure and behaviour. For this reason, the first step (*M1*) decomposes a timethread map into a set of individual timethreads (the `Descriptions` section of the TMDL description) and a timethread topology, i.e., a set of timethread interactions (the `Interactions` section of TMDL).

Structure Interpretation Method (*M2*): The goal of this method is to generate a LOTOS structure expression which expresses the topology of interacting timethreads in a system. The generation of this LOTOS expression is based on a method defined in (Bordeleau, 1993). Other similar methods, where graph structures are transformed into LOTOS expressions (trees), can be found in (Bolognesi, 1990, de Frutos-Eserig, 1993, and Hinterplattner and al., 1993).

Single Timethread Interpretation Method (*M3*): This method aims at generating LOTOS behaviour expressions from single timethreads. Each behaviour expression is a LOTOS process corresponding to its associated timethread.

Specification Composition Method (*M4*): We collect the structure expression and the set of behaviour expressions previously generated to produce a LOTOS specification.

2.2 Analysis and validation in LOTOS

Being formal and algebraic in nature, LOTOS lends itself to validation activities, which often are based on bisimulation concepts. This can help verifying that two specifications at two different levels of abstraction, or having different structures, are indeed comparable. If a LOTOS specification has a finite model, temporal logic analysis, such as model-checking, is possible. Being executable, a LOTOS specification produces a prototype of the entity specified, prototype which can be analyzed and tested (design-level testing). This opens a number of possibilities for validation, of which a few are demonstrated later in this paper.

In our methodology, a formal interpretation model (LOTOS) is obtained from a semi-formal notation (Timethreads). Therefore, the correctness of a timethread-to-LOTOS translation cannot be ensured.

3 EXAMPLE OF A SIMPLIFIED TELEPHONE SYSTEM

3.1 System description

We use the connection phase of a simplified telephone system to illustrate the methodology because most people intuitively know and understand this application. An *Initiator* tries to establish a connection with a *Responder* via a simple *Switch*, and several results, such as a completed connection, a disconnection, a wrong number, a busy tone, etc., can occur. We hereby define the activities that are used in our example as the requirements:

The Initiator:

- *OffHook*: picks the phone up. This is the first activity that initiates the connection phase.
- *HangUp*: hangs up. This activity can occur any time after a *OffHook*. It cancels the connection and disconnects the initiator.
- *Tone*: hears the dial tone. It occurs after *RegInit*.
- *Dial*: dials the phone number of the responder. At this level of abstraction, we assume it to be an atomic activity. It occurs after *Tone*.
- *ErrorDial*: receives an indication telling that an error occurred while dialing. The only option afterwards is for the initiator to hang up. It occurs if *Dial* does not occur in time.
- *WrongNumber*: receives an indication that the number does not exist. The only option afterwards is for the initiator to hang up. It occurs as a result of *CheckDB*.

- *BusyTone*: receives an indication telling that the responder is already using her phone. The only option afterwards is for the initiator to hang up. It occurs as a result of *CheckDB*.
- *NoAnswer*: receives an indication telling that the responder has not responded in time. The only option afterwards is for the initiator to hang up. It occurs if *Answer* does not occur in time

The Responder:

- *Ring*: can hear the phone ringing. It occurs after *CheckDB* if the responder is free.
- *Answer*: picks up the phone to answer the call. It can occur after *Ring*.

The Switch:

- *RegInit*: registers and initializes the user in its database. First activity after *OffHook*.
- *CheckDB*: checks the responder in the database. Indicates whether she exists or not, and whether her phone is already busy or not. It occurs after a correct dialing.
- *Connected*: indicates the expected end of the connection phase. Results from a correct answering.
- *Disconnected*: goes to this state after *HangUp* in connection phase, and updates the database.

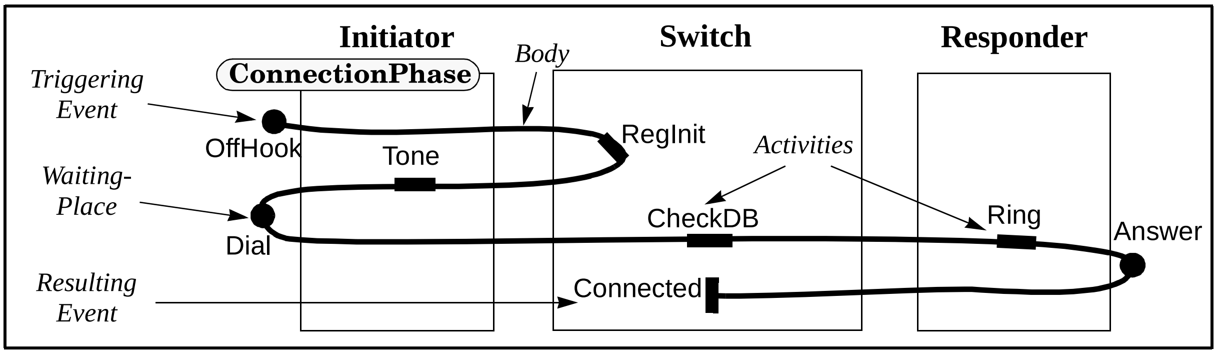
Note that our presentation is at an abstraction level where detailed-design mechanisms are not considered, for instance we do not consider the number of users. Such items can be considered later in the design process.

3.2 Timethread map

In this section, we introduce the Timethread methodology by describing how single timethreads and composed timethread maps are constructed from requirements.

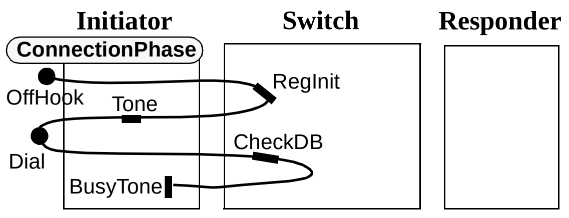
Single timethreads

The notation used to construct single timethreads includes several basic symbols that are identified in the Timethread map 1. Such a typical timethread starts with a *waiting-place* (triggering event ●) and ends with a *junction point* (resulting event ▌). The *body*, on which *activities* are sequentially placed, links the triggering event to its resulting event(s). Notions such as choice, parallelism, time-out and synchronization can also be expressed.

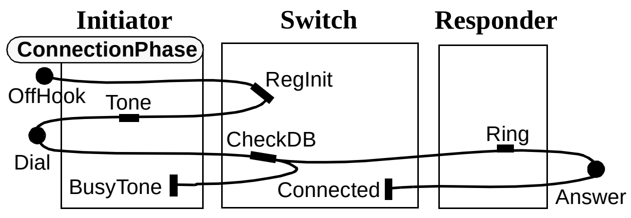


Timethread map 1: Successful connection.

This first timethread presents the traditional scenario path of a successful connection phase. The second timethread (Timethread map 2) illustrates the path where the initiator receives a busy signal, as a result of the responder's phone being already busy. Other timethreads could be derived from the requirements, but we will consider these two only to illustrate the key features of the methodology.



Timethread map 2: Initiator gets a busy tone.



Timethread map 3: Merging of Map 1 and Map 2.

Composed timethread maps

The methodology allows the composition of multiple paths in a single map. The result of timethread composition is either the merging of several timethreads into a single one that offers alternate paths (choice), e.g. Timethread map 3, or the connection of timethreads in a single map, e.g. Timethread map 4. In the latter case, timethreads interact using internal events. Merged and connected timethreads provide new scenario paths resulting from the composition of simpler ones.

Timethread map 3 presents a merging of maps 1 and 2; they have a common sequence of activities and then a choice has to be made based on the system state (information received from *CheckDB*) in order to follow one path or the other.

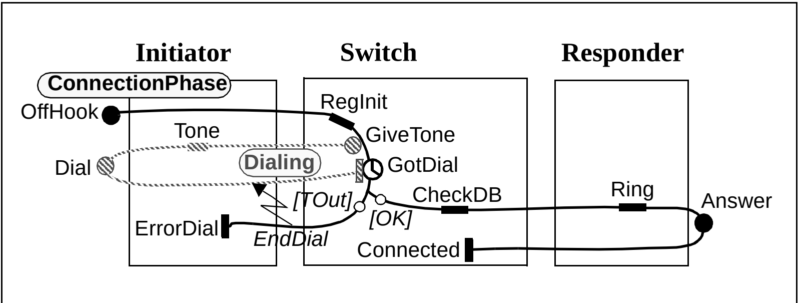
Interactions may occur on waiting-places and junction points. For instance, two synchronous interactions occur between timethreads *ConnectionPhase* and *Dialing* on the waiting-places *GiveTone* and *GotDial* (two new internal activities) in Timethread map 4.

The *in-passing* semantics of interactions is visually represented by the triggering event of a timethread *TT1* being juxtaposed to the body of a timethread *TT2*. When *TT2* reaches this point on the path, it triggers *TT1* and then continues. This is the case for activity *GiveTone* in Map 4. A waiting-place on a body waits for an event to occur from the environment (ex.: *Dial*) or from the result of another timethread (ex.: *GotDial*).

Several types of transformations, such as splitting, extension, and refinement, can be applied on timethread maps. Map 4 is the result of two such transformations applied to Map 1.

The successful connection has been **extended** with a new alternative result (*ErrorDial*), caused by a time-out during the dialing. This time-out occurs at the time waiting-place (🕒) *GotDial*, which then returns *TOut* (a *tag*, or a value) as a result. Note that guards (*[TOut]* and *[OK]*) have been added on the alternative paths. Then, a segment of the path is **split** to allow parallelism between the dialing sequence and the rest of the path starting with the time waiting-place. The timethread *Dialing* is therefore created and finally composed with the original *ConnectionPhase* timethread on two synchronization points.

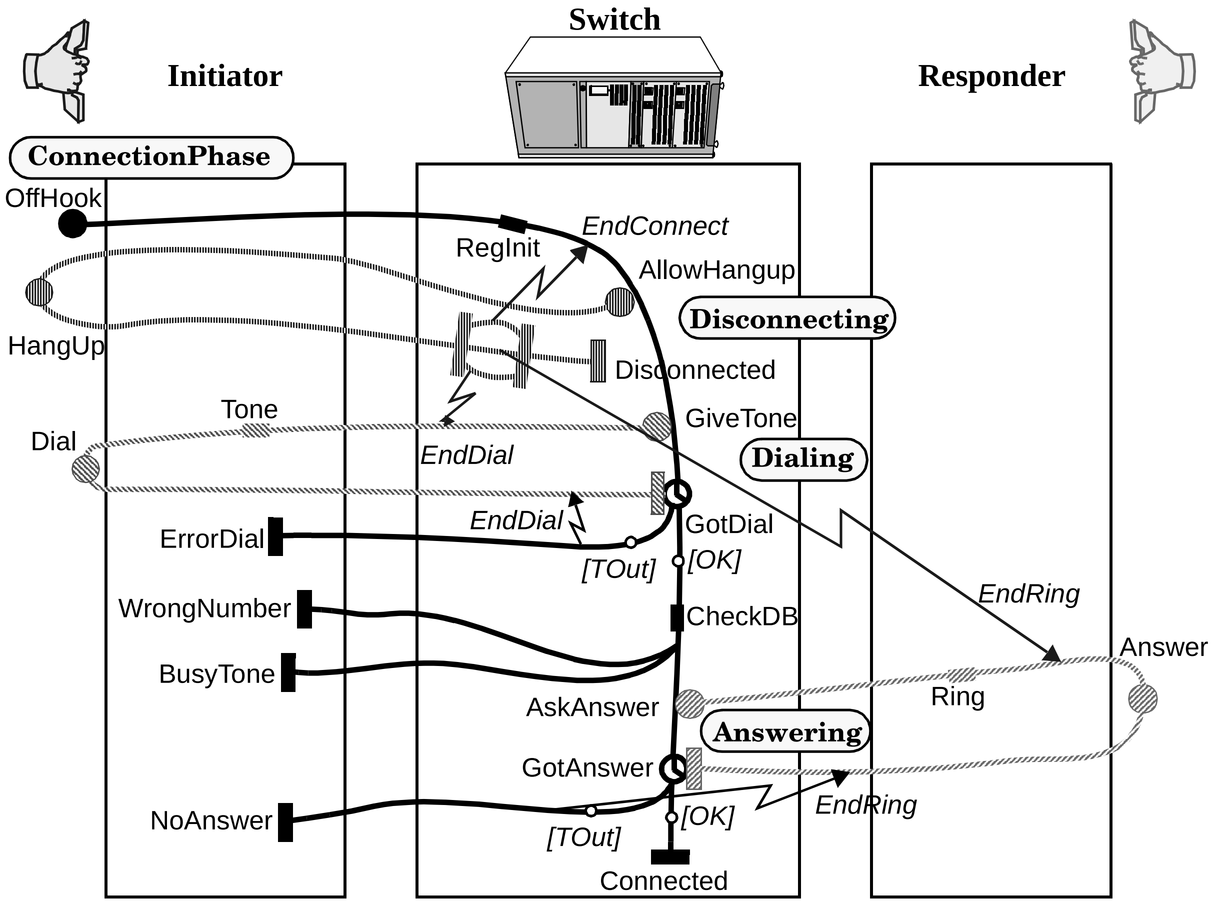
The broken arrow (↘) represents an abort activity, which means that *EndDial* (an internal activity) will destroy or disable ongoing scenarios along the timethread *Dialing*, similarly to a reset feature.



Timethread map 4: Splitting and extension of Map 1.

Final timethread map

Along the way, we could create many single timethreads and compose or transform them into more complex timethread maps, as we did for Timethread map 4. We do so until we reach the point where we have captured all the requirements. The final Timethread map 5 represents the requirements at the level of abstraction chosen for this paper. Although space constraints forbid us to cover here the complete process that led to this map, we raise several points to give the reader a better understanding of the timethread methodology and the operations involved.



Timethread map 5: Simple telephone connection (Final map).

Our design process resulted in four interconnected timethreads:

- *ConnectionPhase*: Gives the initiator the result of the triggering event *OffHook*. Triggers the three other timethreads along the path.
- *Disconnecting*: Manages the *HangUp* activity and then returns to a disconnected state.
- *Dialing*: Manages the initiator's activity *Dial*.
- *Answering*: Manages the responder's activity *Answer*.

To compose these timethreads (synchronously), several new internal events were created:

- *AllowHangup*: Triggers the *Disconnecting* timethread that allows the *HangUp* event to be received.
- *GiveTone*: Triggers the *Dialing* timethread.
- *GotDial*: Indicates that the *Dial* arrived in time. This is a time waiting-place, so a time-out could occur (with the tag value *TOut* as a result).
- *AskAnswer*: Triggers the *Answering* timethread.
- *GotAnswer*: Indicates that the *Answer* arrived in time. This is a time waiting-place, so a time-out could occur (with the tag value *TOut* as a result).
- *EndConnect*: Aborts the timethread *ConnectionPhase*.
- *EndDial*: Aborts the timethread *Dialing*.
- *EndRing*: Aborts the timethread *Answering*.

In the *Disconnecting* timethread, we also find an instance of three activities (*EndConnect*, *EndDial*, *EndRing*) that are performed in parallel.

3.3 Application of the Interpretation Method

Once we have a satisfactory timethread map, we use the LOTOS interpretation method to generate the specification to be analyzed and validated against the system requirements.

As a result of the timethread map decomposition, we obtain a timethread topology such as the one in Figure 3(a), and a set of single timethreads descriptions, such as the one in Figure 4(a). Both come from their respective sections in the TMDL description. We do not show the final map as an example because it involves too many details. Its TMDL description and LOTOS specification are available upon request to the authors.

Note that we can define internal activities both globally in the map and locally in a timethread (see the two examples). For that purpose, we use the LOTOS *hide* operator in the specifications. Any activity can be defined as internal, depending on the level of abstraction.

The second step in the LOTOS interpretation method consists in the application of the structure interpretation method, where a LOTOS structure is generated from the topology of interacting timethreads (the `Interactions` section in TMDL). In Figure 3(b), the result obtained from our compiler is shown. We can observe that timethread interactions are mapped onto synchronization gates in the LOTOS specification, and individual timethreads are mapped onto LOTOS processes.

(a) Map 4 structure: TMDL description

```
Map Map4 is

INTERNAL  { Hidden global activities }
   GiveTone, GotDial, EndDial

INTERACTIONS
   ConnectionPhase, Dialing
      on  GiveTone, GotDial, EndDial;

DESCRIPTIONS
   Timethread ConnectionPhase is
   { Single timethread description... }
   EndTT { ConnectionPhase }

   Timethread Dialing is
   { Single timethread description... }
   EndTT { Dialing }

EndMap { Scenario4 }
```

(b) Map 4 structure: LOTOS specification

```
specification Map4[Answer, Connected, Dial,
                   ErrorDial, OffHook, Ring, Tone]
                   :noexit

library
   Boolean, NaturalNumber
endlib

(* Tag ADT definition *)
type Tag is Boolean, NaturalNumber
   (* Type description... *)
endtype

behaviour

hide EndDial, GiveTone, GotDial in
   (
      ConnectionPhase[Answer, Connected, EndDial,
                      ErrorDial, GiveTone, GotDial,
                      OffHook, Ring]
      |[EndDial, GiveTone, GotDial]|
      Dialing[Dial, EndDial,GiveTone, GotDial,Tone]
   )

where
   process ConnectionPhase[...]:noexit :=
      (* Description... *)
   endproc  (* Timethread ConnectionPhase *)

   process Dialing[...]:noexit :=
      (* Description... *)
   endproc  (* Timethread Dialing *)

endspec (* Map Scenario4 *)
```

Figure 3 Map structure (Map 4) in TMDL and LOTOS.

In the third step (see Figure 4), we apply the single timethread interpretation method to generate LOTOS processes, or behaviour expressions, from their corresponding timethreads (from the `Interactions` section in the TMDL description). Note that triggers, waiting-places, activities (actions) and results all correspond to LOTOS gates. The identifiers used in the LOTOS specification are the ones used in the TMDL description. More complex TMDL operators make use of other LOTOS constructs such as choices, interleavings, synchronizations, disablings, ADTs and guards, recursion, etc. In the fourth step, we combine the LOTOS structure expression and behaviour expressions in a global LOTOS specification. The latter reflects the scenario paths induced by the timethread map, and is thereafter used by the analysis and validation tools.

(a) *ConnectionPhase*: TMDL timethread

```
Timethread ConnectionPhase is
   { This is the main timethread. }
   Internal
        RegInit,
        CheckDB

   Trigger (OffHook)
   Action (RegInit)
   Action (Tone)
   Wait (Dial)
   Action (CheckDB)
   Action (Ring)
   Wait (Answer)
   Result (Connected)
EndTT { ConnectionPhase }
```

(b) *ConnectionPhase*: LOTOS process

```
process ConnectionPhase[BusyTone, Dial, OffHook,
                        Tone]:noexit :=
hide CheckDB, RegInit in

   OffHook;
   (
      RegInit;
      Tone;
      Dial;
      CheckDB;
      Ring;
      Answer;
      Connected; stop  (* No recursion *)
   )
endproc  (* Timethread ConnectionPhase *)
```

Figure 4 Timethread *ConnectionPhase* (Map 1) in TMDL and LOTOS.

3.4 Formal analysis and validation

We now illustrate a number of LOTOS-based techniques and tools to check the high-level design. With available LOTOS tools, we can analyze and test whether the specification accepts all the scenarios from the requirements and refuses undesirable scenarios. We use model-checking and goal-oriented execution to verify properties such as absence of deadlocks and race conditions. We present in this section several key examples related to our case study.

Analysis

The composition of timethreads results in the emergence of new scenarios, sometimes undesirable. The analysis of the specification corresponding to the final composed map helps in discovering these scenarios. LOTOS analysis can be done using step-by-step execution of the specification with tools such as ELUDO (Ghribi and Logrippo, 1993) and LOLA (Quemada, Pavón, and Fernández, 1988). We used LOLA in the following two examples.

Trace 1 in Figure 5(a) shows a global and expected scenario leading to a successful connection. Internal actions starting with a `Sync_` are automatically added by our TMDL-to-LOTOS compiler for internal synchronizations. For instance, `Sync_time_1` checks whether there is a time-out or not.

Trace 2 presents a potential problem discovered in the high-level design. We see that a *Dial* can be followed by an *ErrorDial*, although this is forbidden in the requirements. The error is caused by a race condition at the waiting-place *GotDial*. A time-out occurred right after *Dial* was performed and before *GotDial* was received. Other such problems can be found during the analysis phase.

The problems and issues discovered during the analysis (and also during testing and verification) do not have to be solved at a timethread level. This might be too high a level of abstraction to do so. The goal here is to discover potential causes of problems that must be solved during the detailed-design phase.

(a) Trace 1: Successful connection

```
[  1] - offhook;
[  1] - i; (* reginit *)
[  1] - i; (* allowhangup *)
[  1] - i; (* givetone *)
[  3] - tone;
[  3] - dial;
[  2] - i; (* gotdial *)
[  1] - i; (* sync_time_1 ! ok *)
[  1] - i; (* checkdb *)
[  3] - i; (* askanswer *)
[  3] - ring;
[  3] - answer;
[  2] - i; (* gotanswer *)
[  1] - i; (* sync_time_3 ! ok ! ok *)
[  1] - connected;
```

(b) Trace 2: Race condition while dialing

```
[  1] - offhook;
[  1] - i; (* reginit *)
[  1] - i; (* allowhangup *)
[  1] - i; (* givetone *)
[  3] - tone;
[  3] - dial;
[  1] - i; (* timeout_0 *)
[  1] - i; (* sync_time_1 ! tout *)
[  1] - i; (* enddial *)
[  1] - errordial;
```

Figure 5 Traces resulting from step-by-step execution with LOLA.

Testing

Step-by-step execution is useful for an intuitive discovery of problems, but the global state space is usually too large for this technique to be efficient. One way to reduce this state space is to compose the specification with a test case, hence allowing an exhaustive search for problems in this context.

Design testing allows us to test internal activities of components by making them visible, in a grey-box fashion. One can check whether or not test cases that must be accepted can be executed by the specification, and whether or not test cases that must be rejected are always forbidden by the specification. This is called acceptance/rejection testing. Our methodology allows us to decide which activities should be internal and which should not, therefore leading to a high level of flexibility for the definition of test cases.

Single timethreads defined while constructing the final map can be used as acceptance test cases. These usually are straightforward scenarios that must be found in the composed map. Figure 6(a) presents a successful-connection test case derived, using the single timethread interpretation method, from the timethread in Map 1. We used the testing capabilities of LOLA to validate *Test1* against the specification, and the result was a *MAY PASS* verdict. We can see from Figure 6(b) that 355 scenarios (traces) are possible, from which 20 lead to successes and 335 lead to deadlocks. This result is due to the combinations of possibilities where a time-out occurs (at *GotDial* or *GotAnswer*). Figure 6(c) presents one of these unexpected scenarios that were obtained from LOLA.

(a) Acceptance Test 1: LOTOS process

```
process Test1[Answer,Connected,Dial,
              OffHook, Ring, Tone,
              Success]:noexit :=
(* Comes directly from map 1 *)
   hide CheckDB, RegInit in
      OffHook;
      (
         RegInit;
         Tone;
         Dial;
         CheckDB;
         Ring;
         Answer;
         Connected;
         Success; stop
            (* No recursion *)
      )
   endproc  (* Test1 *)
```

(b) Execution on LOLA

```
lola> TestExpand 100 Success
      Test1 -y -i

  Analysed states       = 1083
  Generated transitions = 1102
  Duplicated states     = 0
  Deadlocks             = 335

  Process Test = test1
  Test result  = MAY PASS.

  355 executions analysed:

              successes = 20
                  stops = 335
                  exits = 0
          cuts by depth = 0
```

(c) Example of Unexpected scenario

```
offhook;
i; (* reginit *)
i; (* reginit *);
i; (* allowhangup *)
i; (* givetone *)
tone;
dial;
i; (* checkdb *)
i; (* timeout_0 *)
i; (* sync_time_1
      ! tout *)
i; (* enddial *)
stop
```

Figure 6 Acceptance Test 1, derived from Map 1.

Test cases can also be generated by hand in LOTOS. For instance, *Test2* (Figure 7) is an acceptance test case checking that connections are impossible when the responder does not perform the *Answer* activity. The result shows that 12 scenarios successfully satisfy the test and that there is no rejection. Therefore the verdict is a *MUST PASS*, as expected.

Rejection test cases are LOTOS processes that must deadlock in all cases when composed with the specification. Such tests could also be defined for our case study.

(a) Acceptance Test 2: LOTOS process

```
process Test2[BusyTone, Dial, OffHook, Tone,
             Success,NoAnswer,ErrorDial]:noexit:=
(* More complex test checking that we cannot be *)
(* connected if the responder does not answer *)
      OffHook;
      (
         (
            Tone;
            Dial;
            BusyTone; Success; stop
                                  (* No recursion *)
         )
         [> (ErrorDial; Success; stop
             []
             NoAnswer; Success; stop)
      )
   endproc  (* Test2 *)
```

(b) Execution on LOLA

```
lola> TestExpand 100 Success Test2 -y -i
    Analysed states       = 52
    Generated transitions = 63
    Duplicated states     = 0
    Deadlocks             = 0

    Process Test = test2
    Test result  = MUST PASS.

                   successes = 12
                       stops = 0
                       exits = 0
               cuts by depth = 0
```

Figure 7 Acceptance Test 2.

Verification using model-checking

Model-checking (Clarke, Emerson, and Sistla, 1986) is a technique for verifying the truth or falsehood of temporal logic properties in a model of the specification. With this technique, we can skip non-essential activities, and thus concentrate on the important ones, in the expression of a property. If test cases are use, instead, all intermediate external activities between two activities of interest must be included.

Some of the temporal logic quantifiers allowed are `AG` (henceforth in all futures), `AF` (eventually in all futures), `EF` (eventually in some future). LMC (the LOTOS Model Checker (Ghribi, 1992)) is a model-checking tool that is part of the University of Ottawa LOTOS toolkit ELUDO. A finite model of the specification is obtained by the tool SELA (also part of ELUDO) and then transformed into a Kripke model by LMC. Some of the design properties checked by using LMC were:

- `EF ( 'Connected' )`: A connection will occur eventually in some future. This is a requirement that should be satisfied by any design. LMC indicates that this formula holds.
- `AG ('Dial' → AF ('WrongNumber'∨'BusyTone'∨'NoAnswer'∨'Connected'))`: All *Dial* will be followed (not necessarily immediately) by one of the four results enumerated. LMC returns a **false** verdict, therefore indicating a problem. There is a race condition at the time waiting-place *GotDial* and the resulting event could be *ErrorDial*. This problem was previously found in the analysis, as illustrated by the second trace of Figure 5.
- `AG ( 'Answer' → AF ('Connected') )`: Every *Answer* will be followed (not necessarily immediately) by a *Connected*. Although this property sounds intuitively correct, LMC's verdict is **false** because of a race condition at the time waiting-place *GotAnswer*, thus avoiding the *Connected* result.
- `EF ( 'HangUp' → EF ('Connected') )`: A *HangUp* may be followed by a *Connected*. This is an example of a negative property that must be rejected by our design. LMC indicates that this formula holds, so we conclude that there is another problem at this level. In this case, a *Connected* could slip between *HangUp* and *EndConnect*, indicating a third race condition.

Many more positive and negative properties derived from the requirements can be verified with model-checking, and other problems than race conditions can be found (non-determinism, wrong ordering, deadlocks, etc.).

Goal-oriented execution

One disadvantage of model-checking is that it requires a complete model of the system to be constructed. For realistic systems, such models are (at best) computationally expensive to obtain. Goal-oriented execution (Haj-Hussein, Sincennes and Logrippo, 1993, and Brinksma and Eertink, 1993) is a LOTOS execution mode that directs execution to see whether certain action sequences can be achieved. Obviously impossible search directions are avoided by our GOAL tool (also part of ELUDO). Here are several design requirements, similar to the properties previously presented using temporal logic, which GOAL allows to verify:

- `GOAL [Connected]`: This goal checks that a *Connected* result can be reached. This gives the set of possible scenarios ending with this activity.

- `GOAL [Dial, ErrorDial]`: This goal checks that a *Dial* can be reached, and that this can be followed by an *ErrorDial.* The requirements state that this situation should not happen. However GOAL indicates that it can happen, because of the race condition at *GotDial*. This goal is complementary to the second temporal logic property to obtain all the traces that lead to *Dial* and then to *ErrorDial*.
- `GOAL [Dial, Answer, NoAnswer]`: This goal checks that a *Dial* can be reached, after this an *Answer* can be reached, and eventually a *NoAnswer* can be reached. This goal outputs the traces that invalidate the third property expressed in the previous section.

Goals are useful by themselves to verify different properties, but they are also powerful as diagnostic tools for problems detected using model-checking.

4 CONCLUSION AND FUTURE WORK

We have demonstrated a design methodology which is based on the use of two complementary techniques: timethreads and process algebras (LOTOS). Timethreads are obtained from scenarios, which in turn are obtained from requirements, and then are combined into high-level designs. Timethread's graphical notation is translated manually into the language TMDL, and this automatically into LOTOS. LOTOS specifications can be analyzed, by using tools, for conformance to requirements. The essential points of this process were demonstrated on a simple telephony example. The validation techniques used were acceptance/rejection testing, model-checking, and goal-oriented execution. In (Amyot 1994), a much larger telepresence example was developed in a similar fashion.

Several research directions need to be pursued. Research must continue towards determining what are the validation techniques that are useful in relation with timethread design, and towards efficiently implementing them. Tools for the methodology must be implemented. In the long range, we see an integrated environment where this type of design and validation techniques can be carried out by users familiar with timethreads only.

5 ACKNOWLEDGMENT

This work was funded in part by the Telecommunications Research Institute of Ontario and by the Natural Science and Engineering Research Council of Canada. We are grateful to Michel Locas, Ron Casselman, Abdellatif Obaid, and Trevor Pearce for contributions that led to this article.

6 REFERENCES

Amyot, D. (1994) *Formalization of Timethreads Using LOTOS*. M.Sc. Thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.

Bolognesi, T. (1990) "A Graphical Composition Theorem for Networks of LOTOS Processes". In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, 88-95.

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

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

Bordeleau, F. and Locas, M. (1994) "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.

Brinksma, E. and Eertink, H. (1993) "Goal-Driven LOTOS Execution". In: A. Danthine, G. Leduc, and P. Wolper (Eds), *Protocol Specification, Testing and Verification, XIII,* North-Holland, 45-60.

Buhr, R.J.A. and Casselman, R.S. (1995) *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA. To appear in October.

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

Clarke, E.M., Emerson, E.A. and A.P. Sistla (1986) "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications". *ACM TOPLAS*, 8(2), 244-263.

de Frutos-Eserig, D. (1993) "A Characterization of LOTOS Representable Networks of Parallel Processes". In: P. Scollo (Ed), *Proceedings of AMAST'93.*

Ghribi, B. (1992) *A Model Checker For LOTOS*. M.Sc. Thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.

Ghribi, B. and Logrippo, L. (1993) "A Validation Environment for LOTOS". In: A. Danthine, G. Leduc, and P. Wolper (Eds), *Protocol Specification, Testing and Verification, XIII,* North-Holland.

Haj-Hussein, M., Logrippo, L. and Sincennes, J. (1993) "Goal Oriented Execution for LOTOS". In: M. Diaz and R. Groz (Eds), *Formal Description Techniques, V,* North-Holland, 311-327.

Hinterplattner, J., Nirshl, H. and Saria H. (1991), "Process Topology Diagram", In: J. Quemada, J. Mañas, and E. Vázquez (Eds), *Formal Description Techniques, III*, North-Holland.

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

Quemada, J., Pavón, S. and Fernández, A. (1988) "Transforming LOTOS Specifications with LOLA: The Parametrized Expansion". In: K. J. Turner (Ed), *Formal Description Techniques, I*, IFIP/North-Holland, 45-54.

Vigder, M. and Buhr, R.J.A. (1992) "Using LOTOS in a Design Environment". In: K.R.Parker, G.A. Rose, (Eds), *Formal Description Techniques, IV,* IFIP/North-Holland, 1-14.

Most theses and technical reports referenced in this paper are available on the World Wide Web at *http://www.csi.uottawa.ca:80/~lotos/ and http://www.sce.carleton.ca/rads/doors.html.*