LOTOS Generation from Timethread Maps: A Language and a Tool

Daniel Amyot

Project report presented to
Luigi Logrippo and
Raymond J.A. Buhr.
CSI 5900

Generating LOTOS specifications from informal timethread maps is an interesting challenge. This projects aims at automating part of the interpretation method introduced in the author's master thesis. We define a textual language (*TMDL*) to represent timethread maps, and a compiler that translates *TMDL* descriptions into LOTOS specifications. The reader of the current report is assumed to have a background on timethreads and LOTOS.

1.0 Introduction

1.1 Tools and the Design Method

The timethread-centered approach [BuC 93] [BuC 94a] helps designers to discover systems functionalities. When a timethread map satisfies the requirements from an informal perspective, we use an interpretation method to generate a description in a given formal method. This formal specification is then used to validate the system designed against requirements, scenarios, use cases or previous design (after some refinements). However, for such method to be useful to designers, different tools must be available.

1.1.1 Timethread Maps

Timethread maps contain possibly more than one timethreads that may interact. They represent causality flows within a system in a visual way. Timethread maps are still informal, and composition rules only start to emerge [Loc 94]. A graphical user interface (*GUI*), or a timethread maps construction tool, needs to be created since none has been programed yet. Such tool would prove very helpful to designers when creating and manipulating timethread maps.

1.1.2 Interpretation Methods

We can interpret timethread maps in many ways. Two partial interpretation methods have already been defined: one gives a Petri nets semantics to timethreads, and the other uses the Formal Description Technique LOTOS as its underlying model. Other interpretation techniques could be invented in order to validate different aspects of a timethread map against the requirements of previous maps. Again, a lack of tools prevents the automated generation of formal descriptions from timethread maps.

1.1.3 Formal Methods

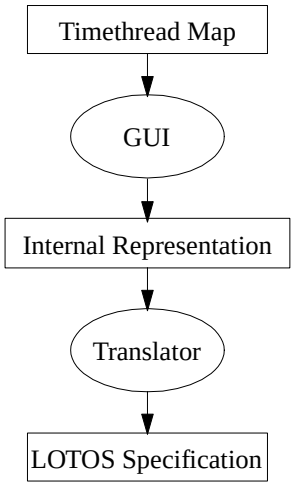
During the last decade, many different formal methods have emerged, and different validation tools were created accordingly. For instance, LOTOS is an algebraic language that includes powerful constructs allowing the definition of a system as a collection of interacting processes. Validation tools for executing, testing and verifying LOTOS specifications are already available from several research groups around the world.

1.2 Problem Definition

A severe lack of tools for the creation of timethread maps and the generation of formal specifications (via *interpretation methods* [Bor 93]) causes the timethread design methodology to be less appealing to system designers. It is necessary to have at least two types of tools to fulfill designers' needs, as presented by the ellipses in figure 1:

- An intelligent GUI, allowing the creation, transformations and manipulations of timethread maps. This tool would be based on a flexible and manageable internal representation. Such a representation, based on hypergraphs, is proposed in [Loc 94].
- Some translator, which translates the internal representation into a specification in a given formal language. This tool in fact an automated interpretation method.

FIGURE 1. High-level view of a translation from a timethread map to a LOTOS specification.



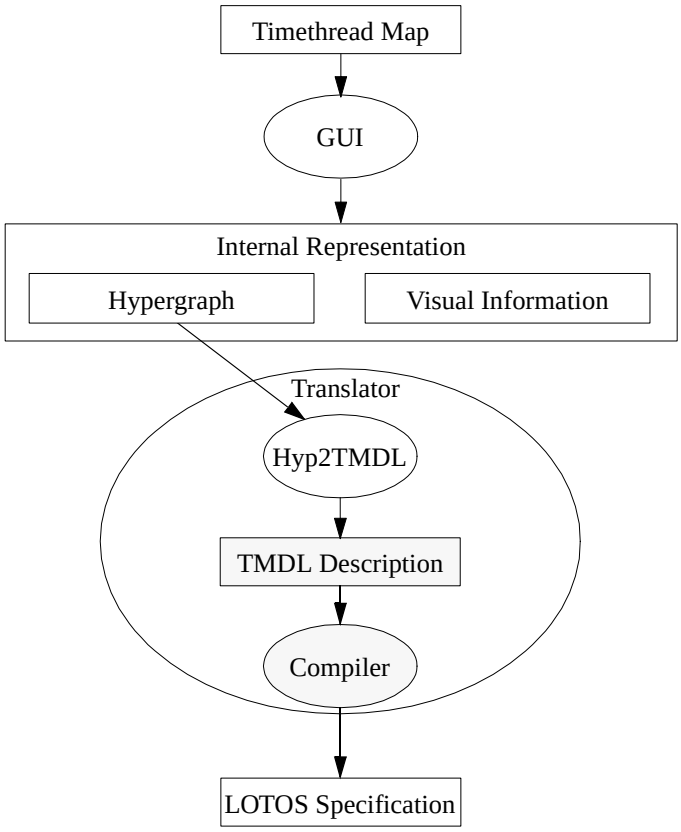
The GUI is a very complex topic by itself, and it will not be discussed here. This report deals with the second type of tool only. More precisely, we will define a textual intermediate representation (called *TMDL*) and an automated LOTOS interpretation method. The creation of a prototype compiler, which generates LOTOS specifications from *TMDL* descriptions, will help us formalize timethreads and the LOTOS interpretation method. We will test this compiler and give some examples, followed by possible enhancements and a conclusion.

2.0 Proposed Approach

2.1 A More Concrete View

We can expand the “Internal Representation” box and “Translator” ellipse from figure 1 to get a more concrete view of the translation procedure proposed (fig. 1). In [Loc 94], the author suggested that the internal representation be composed of *visual information* (location, shapes, colors, size...) and a graph representation based on *hypergraphs*.

FIGURE 2. More concrete view of a translation from a timethread map to a LOTOS specification.



Hypergraphs allow the creation, management, and transformations of timethreads. Could we generate LOTOS specifications from hypergraphs? No doubt that the information needed is there, but we believe it is too high a step to go directly from hypergraphs to LOTOS. We want hypergraphs to remain as close to timethread maps as possible, without too much influence coming from the problems caused by the generation of a specification in some formal language. Also, because this hypergraph representation is still evolving, any change might result in very difficult problems for a highly-coupled translator.

We therefore propose an interface between hypergraphs and LOTOS. It would have to be generated from hypergraphs, but in a format suitable for an eventual LOTOS code generation. This interface is the *Timethread Maps Description Language* (*TMDL*). TMDL has to be close to both timethreads (and hypergraphs) and LOTOS.

Such an interface has the advantage of separating concerns w.r.t. timethreads internal representation and LOTOS code generation. Changes in the hypergraph representation will not affect the way LOTOS specifications are generated, and vice-versa.

As shown in figure 1, we would of course need a tool (called *Hyp2TMDL*) to generate *TMDL* descriptions from hypergraphs (and this must be easier than generating LOTOS directly). However, this tool will not be discussed here because the hypergraph representation is still ongoing work. We will discuss instead the shaded language and the shaded tool from the figure: *TMDL* and a *TMDL-to-LOTOS compiler.*

2.2 The Language

We present here several requirements for this textual intermediate representation of timethread maps. The language we obtain is an extension of *STDL*, presented in the thesis [Amy 94], that will include *LARG* descriptions.

2.2.1 STDL

This *Single Timethread Description Language* was created to represent individual timethreads as entities in their own right. Because it is defined as an EBNF grammar, *STDL* implicitly includes several creation rules, indicating what is a valid timethread.

2.2.2 LARGs

These *LOTOS Architectural Representation Graphs* [Bor 93], now called *SR-Graphs*, are used in the thesis in order to represent interactions between timethreads. Although formally defined as tuples in Bordeleau's thesis, no *LARG* language or tool is available yet.

2.2.3 TMDL

This new language's intent is to include both *STDL* and *LARGs* in one unique model. The *Timethread Map Description Language* is to be defined as a BNF grammar to which a set of external semantic rules will be attached. Issues related to *TMDL* are:

- Creation of a new grammar.
- Definition of constructors representing interactions (timethreads and events), and recursive groupings.
- Specification of rules indicating valid and invalid interactions.

- Inclusion of internal (hidden) and external events, therefore defining the system interface.
- Integration (and perhaps modification) of *STDL* in *TMDL*.
- Keeping in mind the extensibility of the language.
- Taking in account the generality of the language (independent from LOTOS).

2.3 The Compiler

The automated tool would have two functionalities. The first one is a transformation based on the analysis and grouping phases of the *LAEG* (*LOTOS Architectural Expression Generation*) method defined in Bordeleau's thesis. The second functionality consists in compiling the transformed *TMDL* description into a LOTOS specification.

2.3.1 LAEG Method

Some algorithms developed in the *LAEG* method can help generating a LOTOS specification. Since LOTOS possesses binary operators only, we cannot map any graph of interacting components onto LOTOS. Therefore, a reorganization of the interactions between timethreads is often necessary.

We do not have to use the whole *LAEG* method. It is too general for timethread-specific needs:

- The analysis phase could be reduced and adapted to suit timethread-specific problems.
- The general grouping algorithm could be reduced to a deterministic linear or binary grouping algorithm, which is sufficient for LOTOS validation purposes.

These analysis and grouping phases should be automated. The input *TMDL* description could be transformed into a *TMDL* description with binary grouping. However, the *LAEG* method will not be implemented in the current compiler and this will still be work to be done.

2.3.2 Analysis and Code Generation

Once the binary grouped *TMDL* description is available, it has to be compiled into a complete and functional LOTOS specification. Many issues can be raised:

- Lexical, syntactical and semantic analysis.
- Generation of the structure from the interaction part of the *TMDL* description. This was already introduced in the *LAEG* method.
- Generation of LOTOS processes corresponding to single timethreads. The thesis already attacked that problem.
- Management of gate parameters.
- Management of Abstract Data Types (for *tags*): type definition, message passing, tags availability, consistency...
- Management of unique names for additional internal synchronization gates.
- Management of additional sub-processes created for loops or special waiting-places.

2.4 Objectives

Many implementation choices and design decisions, perhaps not always optimal, will have to be taken. The resulting tool is a *prototype* and might not implement all desirable functionalities. However, the compiler prototype will allow:

- A first taste of automated translation of timethread maps into LOTOS.
- To raise new issues related to the formalization of timethreads.
- To improve the interpretation method and *TMDL*.
- To build case studies more easily.
- To show the potential of this design method.

3.0 From STDL to TMDL

3.1 Need for TMDL

STDL is a language built to describe single timethreads. Its possesses constructs to represents activities, timethread constructs (AND-Joins, OR-Forks...), special waiting-places, internal/external activities, tags, etc. However, *STDL* cannot represent entire timethread maps because it lacks constructs to represent interactions between timethreads.

TMDL intends to augment *STDL* with a set of new constructs allowing the representations of whole maps. These simple constructs will allow the generation of a *LARG* from which LOTOS structures can be derived. Also, since we do not plan to implement the *LAEG* method directly in our compiler, *TMDL* will allow groupings (see [Bor 93]) as input. In this way, an additional *LAEG* module could be added later on to produce a grouped *TMDL* structure from an ungrouped one.

3.2 TMDL Sections

A TMDL description is to be structured in different sections: internal (optional), interactions (mandatory), and descriptions (mandatory). Reserved words are in **bold**.

```
Map Map_Id Is
Internal
   ...  { List of internal activities }
Interactions
   ...  { List of timethreads/groups interactions }
Descriptions
   ...  { List of STDL single timethread descriptions }
EndMap
```

In the optional **`Internal`** section, we can place the list of activities internal to the system. They will therefore be unobservable from a user's viewpoint. These activities will become the list of hidden gates in the global *LARG*.

Example: **`Internal`** `Event1, Event2, Event3`

The mandatory section **Interactions** is where we use the new *TMDL* interaction constructs. An interaction is described as a set of timethreads/groupings interacting on a set of events. We can have a list of interactions in this section. Finally, groupings are defined recursively as list of interactions.

Example:

```
Interactions
   TT1, &Gr1, TT2 on Event1, Event2;
   TT3, TT4 on Nothing;
   Where
      Group &Gr1 Is
         TT5, TT6 on Event3;
      EndGroup
```

The **Descriptions** section simply includes the list of single timethreads described in *STDL*. This section is also mandatory.

Example:

```
Descriptions
   Timethread TT1 Is
      ... { STDL description }
   EndTT
   Timethread TT2 Is
      ... { STDL description }
   EndTT
```

3.3 Modifications to STDL

The original STDL [Amy 94] is slightly modified in this report.

- A new rule that describe the timethread level of specification is added. Three different options are available: single instance - no recursion (default), single instance - end recursion, and parallel recursion (multiple simultaneous instances). The rule is defined as: `<R_level> : "NoRec" | "EndRec" | "ParaRec" | ;`
- A timethread can now be aborted by more than one abort event (rule `<R_LIST_aborted>`).
- Waiting places options now also include **Signal** and **Memory** waiting places (rule `<R_wpoptions>`).
- Rules for compulsory and optional loop segments now merged into a unique rule (`<R_loopcompandopt>`).
- The `<R_Constrained>` waiting place option, although still accepted by the language, will not be given any special semantics. The use of level options (`NoRec` and `EndRec`) will be promoted instead.

3.4 TMDL Grammar

3.4.1 From EBNF to BNF

In the thesis [Amy 94], the *STDL* grammar was defined using an Extended Backus-Naur Form (EBNF), where lists (`{...}`) and options (`[...]`) are permitted. However, since the compiler-builder tool *bison* needs a regular BNF context-free grammar as input file, we have to expand lists and options. Table 1 presents the general guidelines used to get BNF rules from EBNF rules.

TABLE 1. From EBNF rules to equivalent BNF rules.

EBNF Rule.	**Equivalent BNF Rules.**
List: <R1> : <R2> {<R3>} <R4> ;	<R1> : <R2> <List_R3> <R4> ; <List_R3> : <List_R3> <R3> ; <List_R3> : ;
Non-emtpy list: <R1> : <R2> {<R3>}* <R4> ;	<R1> : <R2> <List_R3> <R4> ; <List_R3> : <List_R3> <R3> ; <List_R3> : <R3> ;
Option: <R1> : <R2> [<R3>] <R4> ;	<R1> : <R2> <Opt_R3> <R4> ; <Opt_R3> : <R3> ; <Opt_R3> : ;
Choice between parenthesis: <R1> : <R2> (<R3> | <R4>) ;	<R1> = <R2> <Rest_R1> ; <Rest_R1> : <R3> ; <Rest_R1> : <R4> ;

As suggested in *bison* documentation, left-recursion is used to expand lists. Also, bison allows rules that have the same left-hand-side to be regrouped using the separator "`|`". For instance, the two rules `<Rest_R1>` in the previous table could be rewritten as the following single rule (for clarity): `<Rest_R1> : <R3> | <R4> ;`

3.4.2 Resulting TMDL Grammar

The resulting *TMDL* BNF grammar is given in appendix A. It includes all the STDL (equivalent) rules, plus additional ones corresponding to the map definition, the internal section, and the interactions section.

Note that rules start with `R_` and they are in rectangles while reserved words are in ellipses. Rounded box represent lexical tokens returned from the scanner. Two types are used here, and we can define them as regular expressions:

```
IDENTIFIER = [a..z]([a..z][0..9])*

NUMALPHA   = [0..9]([a..z][0..9])*
```

4.0 Building a Compiler

4.1 Terminology

In the compilers-world, many words and expressions have their specific terminology [FLB 88]. We recall the most important ones here:

- *Tokens*: They are the lowest level symbols used to define a programming language syntax (reserved words, integers, arithmetic symbols, punctuation...)
- *Lexical analyzer* or *Scanner*: A function that reads an input stream, character by character, and returns tokens one by one. It also usually eliminates comments.
- *Parser*: A function that recognizes valid sentences of a language by analyzing the syntax structure of a set of tokens returned from a lexical analyzer. It verifies correct syntax w.r.t. the rules expressed in the context-free grammar.
- *Symbol table*: A data structure where symbol names and associated data are stored during parsing to allow for recognition and use of existing information in repeated uses of a symbol.
- *Semantic routines*: They check the static semantics of each construct and then, if no problem is detected, they generate the (internal) code that correctly implements the construct (they give its meaning to a construct).
- *Static semantic*s: Set of restrictions that determine which syntactically legal programs are actually considered valid (identifiers declared, operators are type-compatible, right number of parameters...). We usually cannot express them in a context-free grammar.

4.2 Compiler Tools

Being a prototype, we want our TMDL-to-LOTOS compiler to be as easy to develop as possible. This is why we use compiler-generator tools such as *lex* (or *flex*) and *yacc* (or *bison*). These tools run under UNIX and generate portable (and quite efficient) C code. The current compiler was developed on a PC-486 under *Linux* 1.0 [Wel 93], and then ported to Sun workstations under *SunOS*. In both cases, the GNU C compiler *gcc* was used. Portability to *MS-DOS* with Turbo C is also possible.

4.2.1 Flex

flex is a tool for generating scanners (or lexical analyzers). It is an enhanced GNU version of the well-known UNIX tool *lex* [LeS 75]. The description is in the form of pairs of regular expressions and C code, called rules. *flex* generates as output a C source file, *lex.yy.c*, which defines the scanning routine *yylex()*.

We preferred *flex* (version 2.3) to *lex* because of a useful feature of the former that allows the generation of a case-insensitive scanner. The case of letters given in the *flex* input patterns is ignored, and tokens in the input are matched regardless of case. The matched text given in *yytext* (returned for identifiers) however has the preserved case.

4.2.2 Bison

bison is a general-purpose parser generator from GNU which is upwardly compatible with input files designed for *yacc* [Joh 75], the standard parser generator under UNIX. *bison* converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. C code can also be inserted for semantic routines and code generation. The version 1.18 was used in our case.

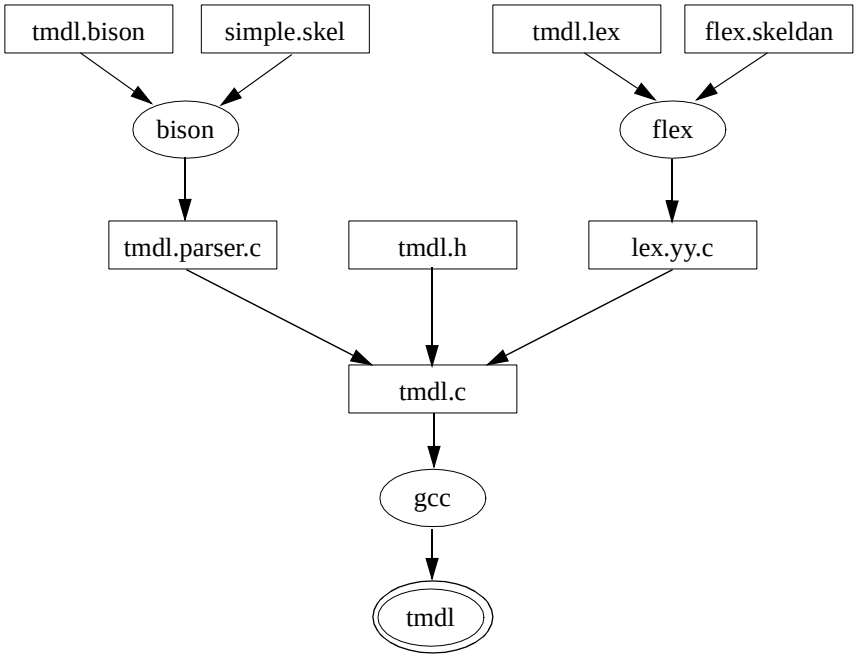
Several bison operators are often used in grammar descriptions:

- *%token*: Declares a terminal symbol (or token).
- *%start*: Specifies the grammar's start symbol.
- *YYSTYPE*: Macro for the data type of semantic values; set to 'int' (integers) by default.
- '*:*' : Separates a rule's result from its components.
- '*;*' : Terminates a rule.
- '|' : Separates alternate rules for the same result nonterminal.
- *yyparse()*: The parser function produced by Bison; call this function to start parsing.
- *yyerror()*: User-supplied function to be called by *yyparse()* on error.

4.2.3 Files Structure

Many different files written in different languages are involved in the making of a compiler. We can see, in figure 3, all the files (rectangles) and tools (ellipses) needed in our case.

FIGURE 3. Files and tools involved in the making of the TMDL-to-LOTOS compiler.



bison needs the grammar file *tmdl.bison* and a skeleton file (*simple.skel*, provided with the tool) to generate the parser function *yyparse()* in *tmdl.parser.c*.

To generate the scanner *yylex()* in *lex.yy.c*, *flex* uses a scanner description (*tmdl.lex*) and another skeleton file (*skeldan.flex*) for general routines. This time, however, we slightly modified the given skeleton file for portability to *MS-DOS*.

Data structures, type definitions and constants are regrouped in *tmdl.h*. *tmdl.c* contains the main function and most of semantic routines. It is used as input to *gcc* to get the final *tmdl* compiler.

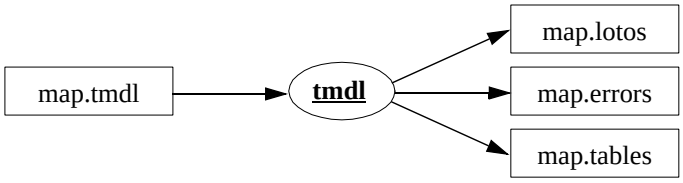
5.0 TMDL-to-LOTOS Compiler

5.1 Functionalities

The TMDL-to-LOTOS compiler, referred as **tmdl** from now on, takes as input a timethread map described in *TMDL* (figure 4). It outputs three different files:

- A LOTOS specification corresponding to the timethread map. It is generated using the interpretation method explained in [Amy 94].
- An error file including all error and warning messages generated during the compilation process. **tmdl** also gives a summary of the number of errors/warnings.
- The different symbol tables used during the compilation. First, the map structure is shown (interaction tree) followed by global symbol tables (timethreads, groups, tag values, internal/external activities). Then, for each timethread in the map, **tmdl** presents the local symbol tables for internal activities, external activities, and tag identifiers. Note that some table elements possess flags that are also displayed (defined, undefined, used, unused, activity type, number of value parameters...). This file is used for debugging purpose only.

FIGURE 4. TMDL compiler functionalities (input and output files).



The user calls **tmdl** in the following UNIX-oriented way:

```
tmdl { <Options> }
```

where the possible list of `<Options>` may include:

- `[-i input_file]` :TMDL description file (default is standard input).
- `[-o output_file]`:LOTOS specification file (default is standard output).

- `[-t output_file]`:Symbol tables file (no file by default).
- `[-e output_file]`:Error and warning file (default is standard error).
- `[-b buffer_size]`:Size of internal buffers, in bytes (default is 8192).
- `[-f]` :Flag for forced output, if an error is encountered (by default, no LOTOS file is output when an error occurs).

The validity of each file is verified, and an error message is emitted when an unknown option is encountered.

5.2 Prototyping

In prototyping a compiler, many important design decisions have to be made. This section enumerates some of the most import ones.

- Use of compiler generators:

In order to build **tmdl**, we chose to make use of usual UNIX compiler generators *flex* and *bison*. They are simple to use, portable, and generate quite efficient code for simple compilers like ours. This choice almost dictates C as the implementation language since the two functions (*yylex()* and *yyparse()*) generated from these tools are C functions.

- Strong coupling:

In a normal compiler, it is useful to have separate modules for parsing, static semantic analysis, and code generation (refer to figure 14 for an example). To get this modularity (or loose coupling), the problem to solve must be very well understood. However, because we were uncertain of what type of problems could arise with our compiler prototype, it seemed simpler to check the static semantic and to generate code directly while parsing (by inserting C code in the grammar in *tmdl.bison*). This strong-coupling solution is common for very simple compilers, but we will see later that we reached the limitations of this approach for **tmdl**.

- Semantic value and buffers:

bison's default semantic value for grammar rule is set to integer, so each recognized rule returns an integer to the calling one. The problem here is that, due to our strong coupling approach, we generate LOTOS code while parsing the input description. The only way to do so is by generating code by parts from the bottom rules to the top rule and therefore a rule must be able to return code to its calling rule. We hence decided to set the semantic value to a pointer to a string (`char *`). To generate the code, we use `sprintf` to print in internal buffers. Three large buffers (their size can be set with the **tmdl** `-b` option), and other small ones (of 1024 bytes) are defined for flexibility. When a code part is complete, the used buffer is duplicated in memory with `strdup` (for the buffer to be reused) and returned to the calling rule which can use it as a parameter to generate its own code part.

Knowing these decisions, we can understand now how each grammar rule is recognized and executed within *bison*. The following steps (figure 5) give a general idea of how we try to insert and to structure C code in each grammar rule:

FIGURE 5. Overview of the steps involved in the recognition and execution of a grammar rule.

1. Set flags (if necessary)
2. Save context on stacks (identifiers, parenthesis, levels...)
3. Set tabulators (if necessary)
4. For each sub-rule/token on the right-hand-side
   - 4.1 Match the sub-rule/token (using bison's engine)
   - 4.2 Get the corresponding code part (associated to parameters such as $1)
5. Retrieve context
6. Analyze static semantics
7. Generate code in buffer (using parameters $1, $2...)
8. Reset tabulators
9. Duplicate buffer
10. Return the duplicated code to the calling rule ($$ = ...).

Note that flags, stacks, and tabulators are discussed later on in section 5.

5.3 Restrictions on TMDL

Due to the complexity coming from the previous decision (strong coupling), some restrictions are made on particular constructs:

- Restriction on loops:

Timethread loops are not allowed to include constructs interpreted as a parallel segment in LOTOS (with operator `|||` or `|[...]|`). Such constructs are `Par`, `Choice`, `OrFork`, `AndFork`, `Async`, `Loss`, and special waiting places (`Delayed`, `Time`, `Signal`). This restriction comes from the decision of implementing the loop in a way different (but simpler) than the one in the thesis. The current loop needs the use of the `exit` LOTOS construct, which creates many problems when combined to concurrency problems (all exits have to synchronize, and this cannot happen).

- Restriction on interactions:

The LAEG method is not part of the current compiler, so the timethread interaction structure needs to be grouped, or *linearized*, in the *TMDL* input description. This means that there must be only one interaction listed at each level (one at the top level, and one per grouping sub-definitions), as in a tree structure. Naturally, an interaction might have more than one member (timethreads or groups) and more than one synchronizing event.

- Restriction on tags:

Our compiler allows the definitions of tags. The management of tags over synchronizing processes is however a complex task, mainly because of LOTOS semantics and scoping rules. Synchronizing processes or behaviours need for each synchronizing event to have the same number, order, and sort of value parameters (luckily, we only use one sort named *Tag*). This consistency has to be dealt with when we generate the specification. Right now, this task is not completed. The main restrictions are therefore to define every tag before using them (with the `Tag` construct), and not to define new tags in loops and special waiting places (with the `?` construct).

- Restriction on special waiting places:

Beside avoiding to use them in loops and not defining new tags within them, there is a last restriction applied to special waiting places. Memory waiting places are not yet available, but they are replaced with normal waiting places instead.

5.4 Data Structures and Related Functions

5.4.1 Constants

Many constants are defined in *tmdl.h* to generalize the data types and data structures definitions. They also enhance the understanding, management and consistency of these types and structures. Such constants are defined for:

- Symbol types in symbol tables (TYPETT, TYPEGROUP, TYPETAGVAL...)
- Symbol attributes in symbol tables (DEFINED, USED, ACTABORT, ACTSYNC..)
- Timethread levels of specification (NOREC, ENDREC, PARAREC)
- Types of waiting places (WPNORMAL, WPDELAYED, WPTIME...)
- Values returned by symbol tables management functions (INSERTED, SET, ...)
- Number of spaces per tab for tabulation (TABSPACE, MORETABS, LESSTABS)

Several constants also constrain to a certain extent what is accepted as *TMDL* input code:

- Maximum depth for paths within paths and other stacks (MAXDEPTH is set to 64)
- Maximum number of members in an interaction (MAXMEMBER is set to 32)
- Maximum number of characters in an identifier (IDMAX is set to 32)

5.4.2 Identifiers

Identifiers are string of IDMAX characters. We also define a structure named id which is composed of:

- `char Orig[IDMAX+1]` : Original symbol identifier.
- `char Low[IDMAX+1]` : Symbol identifier, in lowercase.

`Orig` is used for code generation while `Low` is used for comparisons when a symbol is inserted or retrieve in a symbol table, to speed-up the searching process (the compiler is case insensitive, but output the original symbol uppercase/lowercase).

Note that the '`+1`' in the array relates to strings representation in C; an additional NULL character is needed to end a string.

Two functions in *tmdl.c* relate to identifiers:

- `void lower (Dest, Source)`: Puts in `Dest` the lowercase string corresponding to `Source`.
- `char * trunc(IdToCheck)` : Truncates identifiers longer than IDMAX characters.

5.4.3 Symbol Tables

The symbol tables used in our compiler are ordered chained lists with a structure (named `symtab`) found in *tmdl.h*:

- `id Id` : Symbol identifier.
- `int Defined, Used` : Symbol attributes.
- `int Line` : Line number of first occurence (for error/warning messages).
- `struct symtab *Next` : Link field (pointer to next list element).

Global symbol tables are used for timethreads (`TTTab`), groups (`GroupTab`), tag identifiers (`TagIdTab`), tag values (`TagValTab`), and activities (`ActTab`).

An extensive set of functions managing these tables are defined in *tmdl.c*:

- `int putsym (SymTable, SymId)` : Adds an element to a symbol table (ordered list).
- `symtab *getsym (SymTable, SymId)` : Returns a pointer to an element in a symbol table.
- `int setsym (SymTable, SymId, SetDefined, SetUsed)` : Sets an element's attributes in a symbol table.
- `void clearsym (SymTable)` : Clears a symbol table.
- `void printsym (SymTable, TableType)` : Formats and prints a symbol table (to the table file).
- `void printallsym ()` : Formats and prints all symbol tables (to the table file). Uses `printsym`.

Symbol tables for external activities, internal activities, and tag identifiers are also defined for each timethread in the *TMDL* description. We use them for static semantic analysis and for the generation of the symbol table file. They are grouped in a list structure, named `localtt`, in the following way:

- `char *LowTTId` : Timethread identifier (lowercase only).
- `symtab *Ext` : Timethread local external activities.
- `symtab *Int` : Timethread local internal activities.
- `symtab *Tag` : Timethread local tag identifiers.
- `struct localtt *Next` : Pointer to next timethread in chained list.

5.4.4 Map Structure

Since we require timethread interactions to be linearized, a simple tree structure is sufficient to represent them (the tree width is MAXMEMBER at most). Thus, the `mapstruct` structure is defined as:

- `id Id` : Timethread or group identifier.
- `int Type` : Node type (TYPETT or TYPEGROUP).
- `symtab *Interactions` : List of synchronization events (NULL when `Nothing`).
- `int NumMembers` : Number of members in group (0 when it is a timethread).
- `struct mapstruct *Member[MAXMEMBER]` : Group members list (node children). Each node is limited to MAXMEMBER members (and children).

Three functions, found in *tmdl.c*, manage the map structure:

- `int putmem (MapStruct, SymId, SymType)` : Places a new member (timethread or group) in the map tree structure.
- `mapstruct *getmem (MapStruct, SymId)` : Returns a pointer to a member in the map structure.
- `void printmem (MapStruct, ParTab)` : Formats and prints a map tree structure (in the table file).

5.4.5 Stacks

Some grammar rules require context saving. Different stacks were used to do so:

- `int Levels[MAXDEPTH]` : To manage Results from rules R_firstpath and R_orfork.
- `int ParCount[MAXDEPTH]` : To save the number of open parenthesis (in tag definitions, guards, forks...).
- `char * UniqueId[MAXDEPTH]` : To save unique identifier temporary variables (e.g., for a And-Fork within a And-Fork).
- `char OldTabs[4*MAXDEPTH]`: To save current tabulation.

No functions were created; four global variables were incremented/decremented instead.

5.5 Static Semantics

Static semantic routines intend to detect as many possible semantic problems as possible in the timethread map, whatever GUI tool there is on top of **tmdl** or whatever LOTOS tool used afterwards. Trying to detects problems here will either validate the output of a GUI tool, or prevent a LOTOS tool from analysis an incorrect specification.

Note that each function related to static semantic starts with "`sem`" and that semantic analysis sections in the source code are identified and commented.

5.5.1 Generic Functions

Before getting into real semantics, we should present the two generic functions used to output all errors and warnings.

- `void semwarning (Category, SymId, Message)` : Outputs a given warning message and increments counter 'warningnum' for the final summary report. The format is: `"WARNING (line %d): %s '%s' %s\n", linenum, Category, SymId, Message`
- `void semerror (Category, SymId, Message)` : Outputs a given error message and increments counter 'errornum' for the final summary report. The format is: `"ERROR (line %d): %s '%s' %s\n", linenum, Category, SymId, Message`

Usually, an error leads to an invalid LOTOS specification while a warning could lead to a valid LOTOS specification but a semantics possibly different than the one intended in the timethread map.

5.5.2 Parsing

If an error occurs during parsing, *bison* makes a call to `yyerror(s)` for error handling. This function then calls: `semerror("Symbol", yytext, "caused a parse error.")` where `yytext` is the last symbol read. No error recovery is intended.

5.5.3 Symbols

Symbols and especially identifiers can be overloaded with many types, could be reserved words, or could lead to static dataflow analysis problems. We use three functions (found in *tmdl.c*) to detect these problems.

- `int semchecksym (SymType, SymId)` : Checks whether or not a symbol is already in other symbol tables. The diffent types of symbols are are treated differently.

  It has been decided that an error is output when:
  - A value identifier is also a timethread, tag, or activity identifier,
  - Any type of identifier is the same as the map identifier.

  A warning occurs when:
  - An activity identifier is also a timethread or tag identifier,
  - A timethread identifier is also a tag identifier.

  The approach taken offers much flexibility in these aspects, and other combinations could be considered.

- `char * checklotosword(IdToCheck)` : Checks an identifier against the list of LOTOS reserved words (`char * LOTOSwords[]`). Common TMDL keywords (*Choice*, *Is*, *Par*, and *Where*) are excluded from the list because the parser already takes care of such identifiers.

  When a LOTOS identifier '*idlotos*' is detected, it is renamed to '*ID_idlotos*', and a warning message is output. No error is necessary since the renamed identifier cannot interfere with other ones (the underscore forbidden in *TMDL*).

- `void semcheckdefuse (SymType)` : Checks that defined symbols in a given table are used, and that used symbols in a given table are defined (static dataflow analysis). The different types (timethreads, groups, tag identifiers, and tag values) of symbols are treated differently, and error messages output accordingly (warnings only for tags).

5.5.4 Interactions

Timethread interactions also have to be validated. When the map structure is parsed, the following static semantic routines are applied:

- `void seminteractions (MapStruct)` : Checks all interactions in a map tree structure. Calls `semcheckttint` on each nodes. Errors are output when:
  - A timethread cannot be found in the description section,
  - One of the synchronizing event is not found in an interacting timethread.

- `void semcheckttint (IntList)` : Checks interactions between timethreads involved in an interaction. It uses a collection of boolean flags to check the presence and type of activities involved in the interaction. These flags are set by checking the corresponding activity type stored in timethreads local symbol tables.

  Warning are output when:
  - The number of tag parameters are not the same for all activities (the LOTOS processes could not synchronize),
  - An abort event is detected without an aborted event,
  - A waiting place interacts with a trigger,
  - An action interacts with other activities,
  - A trigger interacts with a trigger,
  - A sync event interacts with activities different from syncs and asyncs.
  - The interaction is composed of asyncs and results only.

  Errors come from:
  - Abort or aborted event interacting with other types of activities.

Again, the approach used is much flexible and allow different semantic rules to be easily created.

5.5.5 Loops

The `Compulsory` and `Optional` sections of a `Loop` must not contain any asynchronous event (Async, AndFork, Par, ...). This leads to a problem with the LOTOS `exit` used in our mapping:

```
(exit ||| stop) ~ stop
```

Because parallel processes would never synchronize on `exit`, the functionality of the `Loop` process, defined as `exit`, should become `noexit` and the loop would never terminate. This is detected by LOTOS tools such as TOPO and XELUDO.

To detect this particular problem in **tmdl**, we check that we do not add a parallel operator (`|||` or `|[...]|`) while creating the loop. If such an operator is found, then an error message is output.

5.5.6 Internal Activities

Some activities are by definition external to timethreads, e.g., triggers, results, syncs, asyncs, abort... Therefore, we cannot make then locally internal. When this happens, a warning is emitted (saying that such activity should not be internal), and the activity is then considered as external.

5.6 Code generation

Explaining 3000 lines of C code is not an easy task, especially when it comes to code generation routines. We assume here that the reader already understands the semantics of the mapping from TMDL constructs to LOTOS constructs covered in [Amy 94]. In this section, we will try to cover the generic routines used to solve issues (enumerated in the thesis) related to lists of gates, ADT generation, loop extra process, complex waiting places, interactions, unique identifiers, dangling parenthesis, levels of specification, and tabulation (for pretty-printing).

5.6.1 Solving Difficult TMDL Constructs Issues

As explained in section 5.2, we decided to generate LOTOS code directly when parsing the input *TMDL* description. However, this does not forbid us to create generic and modular functions to accomplish this task, on the contrary. Here are the most important ones:

Lists of Gates

- `char * makegates (SymTable)` : Returns the list of all gates from a symbol table. Used to generate LOTOS processes formal and actual parameters.

- `char * makeinternal (SymTable)` : Returns the list of internal activities from a symbol table. Used when generating the LOTOS *hide...in* construct.

- `void makespecgates()` : Places the list of map external activities in a symbol table. Computes this list from all the timethread external gates that are not globally internal.

ADTs

- `char * maketaglist (SymTable, SendReceive)` : Makes the list of tags, separated with ? for send and ! for receive, from a symbol table. Used for triggers, results, waiting places, etc. Internal LOTOS synchronization gates also use this function to pass data values from one scope to another.

- `char * makeadt()` : Creates the ADT Tag definition at the beginning of the specification. The `sort Tag` is created and mapped onto natural numbers (from the LOTOS library) with an equation of the format `N : Tag -> Nat`. A dummy tag value is always created first (`dummy_val`), and then all the tag values from the ordered symbol table are mapped onto naturals. Two equations over tags are provided: `eq` for equality and `ne` for inequality.

Loop

- `char * makelooppar (SymTable, Type)` : Makes the loop parameters for tags from a symbol table. When the loop is first created, formal and actual parameters for gates and values are unknown and are therefore replaced with symbols (forbidden in TMDL to avoid side-effects):
  - '@' for actual parameters,
  - '+' for formal parameters,
  - '#' for formal parameters between parenthesis,

- '%' for exit parameters,
- '^' for accept.

When all the information is available, these symbols are replaced with their corresponding parameter list (by using function `subst`).

- `void subst (Dest, Source, CharFrom, StringTo)` : Generic function that replaces a character (`CharFrom`) from the `Source` string with `StringTo` and places the result in `Dest`.

Abort

- `void makeabort(Buf)` : Completes an aborted timethread LOTOS process, when necessary, by adding the disabling event (`[> AbortEvent; ...`) at the end of the process.

Waiting Places

- `void makewait(Buf, SymWPType, SymId, SymPar)` : Creates complex waiting places from `SymWPType`. It generates new tag identifiers and values for `Time` and `Signal` waiting places, and manages the internal synchronizations. `Memory` waiting places are not implemented yet, but this generic process would allow to do so.

Interactions

- `int makeintlist(IntList, MapStruct, SymId, IntDepth)` : Creates the activity interaction list corresponding to the map structure (or a sub map in the global tree). Used to synchronize LOTOS processes (`TT1 |[intlist]| TT2`).

- `char * makebeh(MapStruct, ParTab)` : Creates the LOTOS behaviour corresponding to the map structure. Groups are replaced with simple parenthesis, and tabulation is managed.

5.6.2 Unique Identifiers

Unique identifiers are necessary when creating new internal synchronization events (`SyncPar_0`, `SyncChoice_2`, ...) and new sub-processes (`Loop_1`, `Loop_3`, ...). To ensure each of these identifiers is unique, we use a counter (`UniqueNameGen`), initialized to 0, and the following function:

- `char *namegen (Name)` : Creates a unique name identifier from parameter `Name`. The returned identifier is the concatenation of `Name`, an underscore (to avoid interference with users identifiers), and the value of `UniqueNameGen`. The counter is incremented at each call.

5.6.3 Dangling Parenthesis

Because we have a highly-coupled 1-pass compiler, it is not always possible, for some constructs such as `Guard` and Tag (mapped onto LOTOS guard and `Let`), to know in advance where a parenthesis should be close. The best we can do is to open the needed parenthesis (as in "**(** `[Guard] -> ...` *`rest of path`*"), remember that we have a dangling parenthesis, and then close it later on.

In our prototype, we use a stack of open parenthesis counters (`int ParCount[]`) that is updates each time a parenthesis is open. A stack is needed for paths within paths. When a *TMDL* path is finished (a `Result` is encountered), the following function is called:

- `void closepar(Code)` : Adds dangling closing parenthesis ‘)’ at the end of a path or sub-path. Manages tabulation and adjust the stack accordingly.

5.6.4 Levels

TMDL allows the use of three important levels of specification, for each timethread individually:

- Single instance, no recursion (`NoRec`),
- Single instance, tail recursion (`EndRec`),
- Multiple instances at once (parallel recursion `ParaRec`).

This influences how the LOTOS process ends. When the timethread’s `Result` is encountered, the code generated between the result and the last parenthesis is:

- When `NoRec` : `stop`
- When `EndRec` : `TimethreadId [>]`
- When `ParaRec` : `stop ||| TimethreadId [>]`

where ‘>’ is replaced with the process list of parameters at a later stage.

5.6.5 Tabulation

For a LOTOS specification to be readable, it must be indented in some way that reflects the structure. Tabulators are used in our case, in every code generation rule, for pretty-printing:

- `TABSPACE`: (Constant) Number of space characters in a tab, found in *tmdl.h*.
- `ATab`: Tab with `TABSPACE` spaces.
- `Tabs`: (string) Tabulation of the current line.
- `void adjusttab(Command)` : Adjusts the number of leading tabs for indentation.
  If `Command` is `MORETABS`, adds a tab to `Tabs`.
  If `Command` is `LESSTABS`, removes a tab from `Tabs`.
- `OldTabs[]`: Stack of tabulator strings (to keep context).
- `TabDepth`: Pointer to the current old tab string in `OldTabs` stack

Because we manage tabs during parsing, many adjustments that might look spaghetti-like are needed. Also, loops tabulations are incorrect due to our now famous strong-coupling approach.

6.0 Testing and Example

One of the major problems with testing a program such as a compiler is that we would have to be exhaustive in order to cover its functionalities adequately. In this section, we present how were tested the grammar rules and semantic routines, and we give an example of a complete compilation (the *Traveler System*).

6.1 Testing Grammar Rules

The *TMDL* syntax was tested during the compiler implementation. The tests are not enumerated here because they are simple, numerous and not very interesting. However, here is a short description of what has been tested.

6.1.1 Scanner

The scanner (generated by *flex*) determines tokens from the input description:

- All *TMDL* keywords have been tested and recognized (even `Comp`, `Compulsory`, `Opt`, and `Optional`).
- Identifiers and numalpha are also correctly recognized. If they have more than `IDMAX` characters, then they are truncated to `IDMAX` characters with a call to `trunc()` before being returned to the parser. `checklotosword` is also applied to rename LOTOS identifiers when necessary.
- Comments (between '`{`' and '`}`') are correctly recognized and ignored during lexical analysis. However, a comment must not go over more than one line, and comments cannot include other comments.
- Tabs, newlines and spaces are correctly skipped.
- Symbol case-insensitivity tested.
- Forbidden symbols (~, @, #, $, %, ^, *, -, +, _, |, \, [,], <, >, ., /, }, :, ', ", ') are detected and they cause a parse error.

6.1.2 Parser

The parser (generated by *bison*) recognizes the grammar rules, especially because the rules are defined in BNF and then directly used in *tmdl.bison*. We tested that:

- All individual rules are correctly parsed.
- Syntax errors are detected at any point.
- TMDL reserved words cannot be interpreted as identifiers.

6.2 Testing Semantic Routines

Semantic routine are a lot more interesting to test. However, an exhaustive testing would be very expensive in our case, even if our prototype compiler is not very large. Therefore, we will simply present a test *TMDL* description for interactions, and another *TMDL* description to test the language's complex operators.

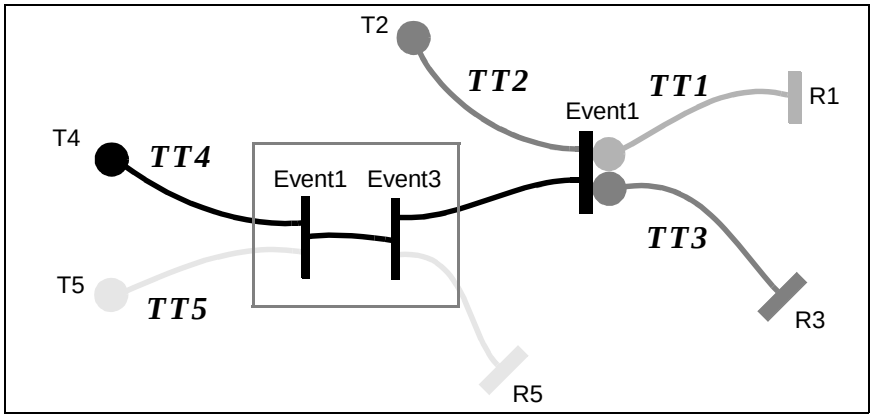
6.2.1 Static Semantics

The errors and warnings presented through sections 5.5 and 5.6 were tested and generated from multiple short test *TMDL* descriptions. They are too numerous to be recalled again here. No problem with the detection methods used seemed to happen.

6.2.2 Interactions

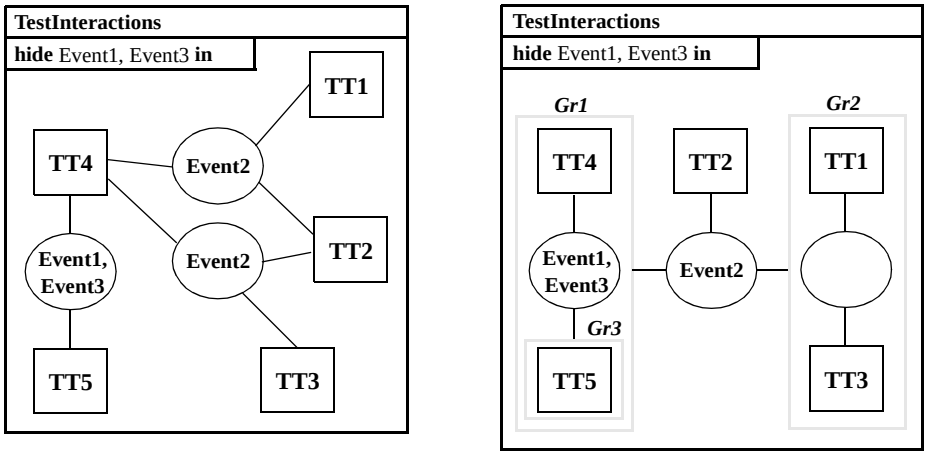
Code generation for interactions is particularly new in this project. To test its correctness, we can use the following timethread map (figure 6). Note that `Event1` and `Event3` (in the dotted box) are internal events.

FIGURE 6. Timethread map *TestInteractions* for interaction testing



We can get a LARG corresponding to these five interacting timethreads (fig.7a), and a linear grouped LARG (fig. 7b) after applying the LAEG method [Bor 93].

FIGURE 7. Original LARG and grouped LARG of *TestInteractions*



(a) **(b)**

The *TMDL* representation of these interactions would be:

```
Map TestInteractions Is

{ This TMDL description comes directly from the original LARG. }
{ It cannot be used as-is by our compiler. Groupings are needed. }

Internal
   Event1, Event3               { 2 internal events }

Interactions
   TT1, TT2, tt4 on Event2;
   TT3, TT2, tt4 on Event2;
   TT4, TT5 on Event1, Event3;

Descriptions
 ... { Timethread TT1 to TT5 descriptions, in any order. }
```

The problem here is that interactions are not linear, i.e., not directly representable in LOTOS. Using the LAEG method, one could form the grouping in figure 7b. Our compiler does require this grouping since the LAEG method is not implemented yet.

A second *TMDL* interactions description for this new LARG is required. The chosen grouping is interesting because it allows us to test internal events and many interaction parameters (number of groups and events).

```
Map TestInteractions Is

{ This TMDL description tests the internal and interactions sections. }
{- Internal and external events }
{- Interactions with 1, 2, and 3 members }
{- Interactions with 0 (Nothing), 1, and 2 events }
{- Groups enumerated in wrong order }
{- Group within a group }

Internal
   Event1, Event3               { 2 internal events }

Interactions
   &Gr1, TT2, &Gr2 on Event2; { 3 members. 1 event }
   Where
      Group &Gr2 is           { Group Gr2 before Group Gr1 }
         TT1, TT3 on Nothing; { 2 members, 0 event }
      EndGroup { &Gr2 }

      Group &Gr1 is
         TT4, &Gr3 on Event1, Event3;   { 2 members, 2 events }
         Where
            Group &Gr3 is     { Group within a group }
               TT5 on Nothing;{ 1 member, 0 event }
            EndGroup { &Gr3 }
      EndGroup { &Gr1 }

Descriptions
 ... { Timethread TT1 to TT5 descriptions, in any order. }
```

Using our compiler, we obtain a LOTOS specification partially reproduced here. No compilation errors or warnings were detected. As you can see, this specification reflects correctly the internal events and map structure in a LOTOS form, and the gate parameters are also correct. Tabulation makes this part easier to read.

```
specification TestInteractions[Event2, R1, R3, R5, Tr2, Tr4, Tr5]:noexit

   ... { Libraries and tag ADT }
behaviour

hide Event1, Event3 in

   (
      (
         TT4[Event1, Event2, Event3, Tr4]
         |[Event1, Event3]|
         (
            TT5[Event1, Event3, R5, Tr5]
         )
      )
      |[Event2]|
      TT2[Event2, Tr2]
      |[Event2]|
      (
         TT1[Event2, R1]
         |||
         TT3[Event2, R3]
      )
   )

where
   ... { Process TT1 to TT5 descriptions }
endspec (* Map TestInteractions *)
```

In the symbol table file generated by our compiler, the map structure is also reflected in the following way:

```
MAP STRUCTURE:
   TestInteractions      TYPEGROUP  3 Event2
      Gr1                   TYPEGROUP  2 Event1, Event3
         TT4                   TYPETT     0
         Gr3                   TYPEGROUP  1
            TT5                   TYPETT     0
      TT2                   TYPETT     0
      Gr2                   TYPEGROUP  2
         TT1                   TYPETT     0
         TT3                   TYPETT     0
************************************************************************
```

6.2.3 Complex Constructs

Testing all *TMDL* constructs is a huge (and tedious) task. In this section, we will test complex constructs only. Simple constructs (`stub`, `trigger`, `result`, `action`, `sync`, `segstub`, and `wait`) will not be discussed as such, mainly because they are usually mapped onto a simple LOTOS gate.

The approach taken here is to regroup all constructs to test into a single (and unrealistic) *TMDL* map description presented in appendix B.1. In this way, we can test multiple constructs at once, reducing the number of timethread processes. The LOTOS specification (file *tests.lot*) is in appendix B.2, and appendix B.3 is the symbol tables (file *tests.tab*).

Here are the test descriptions and results. It is suggested to read the *TMDL* description and its corresponding LOTOS process at the same time.:

- **Abort and Levels**: Timethreads TestAbort, TestAborted1, TestAborted2.

These timethreads test aborts and levels of specifications. We see how the disable operator is used in LOTOS processes to implement `AbortedOn`. The three levels of specification (NoRec, EndRec, and ParaRec) have also been tested, and we can see the impact on the recursion in the LOTOS processes.

- **Choice, Async, Tags, and Guards**: Timethread TestChoice.

This timethread tests a `Choice` within a `Choice`. It also uses the `Async` activity and boolean operators over guards. Also, tag passing over internal synchronizations (for the two `Choice`) is tested. In the LOTOS process, the choices are correctly specified and two unique internal synchronization events (`Sync_Or_0` and `Sync_Or_1`) are created. The tag `Cond` used is passed over the two synchronizing events to make it accessible to the rest of the process. Guards, boolean, and equality operators are well translated too. Finally, the asynchronous event `Act3` is mapped onto an interleaving sub-behaviour.

- **Par, Async, and Tags**: Timethread TestPar.

This timethread tests a `Par` within a `Par`. It also uses the `Async`, and tag passing over internal synchronizations (for the two `Par`) is tested. In the LOTOS process, the parallel segments are correctly specified and two unique internal synchronization events (`Sync_And_2` and `Sync_And_3`) are created. The tag `Cond` used is passed over the two synchronizing events to make it accessible to the rest of the process. The asynchronous event `Act3` is again mapped onto an interleaving sub-behaviour.

- **Loops, Tags, and Guards**: Timethread TestLoops.

A `Loop` within a `Loop` is tested in this timethread. Tags and guards are used to test tag passing from and to loop sub-processes. We also use short and long keywords for `Compulsory` and `Optional` segments. Within the LOTOS process, two loop sub-processes are created (`Loop_4` and `Loop_5`). Tag and gate parameters are correctly listed and managed when entering loops, and when exiting loops (with the `>> accept...in` construct). Of course, no activity leading to LOTOS parallel operators were used in the loops, because this would have led to an error.

- **Special waiting places**: Timethread TestWP.

Here we test waiting place options `Time`, `Delayed`, and `Signal`. The timed trigger causes two new internal events to be created (`TimeOut6` and `Sync_Time_7`). They are used to decide the verdict (values `TOut` or `OK`) passed to the new tag `TimeT8`. The delay of the delayed waiting place is represented with the simple new internal action `Delay_8`. Finally, the signal waiting place causes the creation of an internal synchronization event (`Sync_Signal_9`). Its result (`Yes` or `No`) is passed to the new tag `SigAct13`. Again we can see that the previous tag (`TimeT8`) was also correctly passed over the synchronization event.

In this quite impressive test, we can see from the LOTOS specification seems to manage in a working way the map structure building, the tag ADT generation, dangling parenthesis, unique identifiers, gates lists, parameters lists, levels of specification, and tabulation (except for loops). Therefore, we can have some trust in the solutions enumerated in section 5.6 for these complex problems.

6.3 Traveler System Example

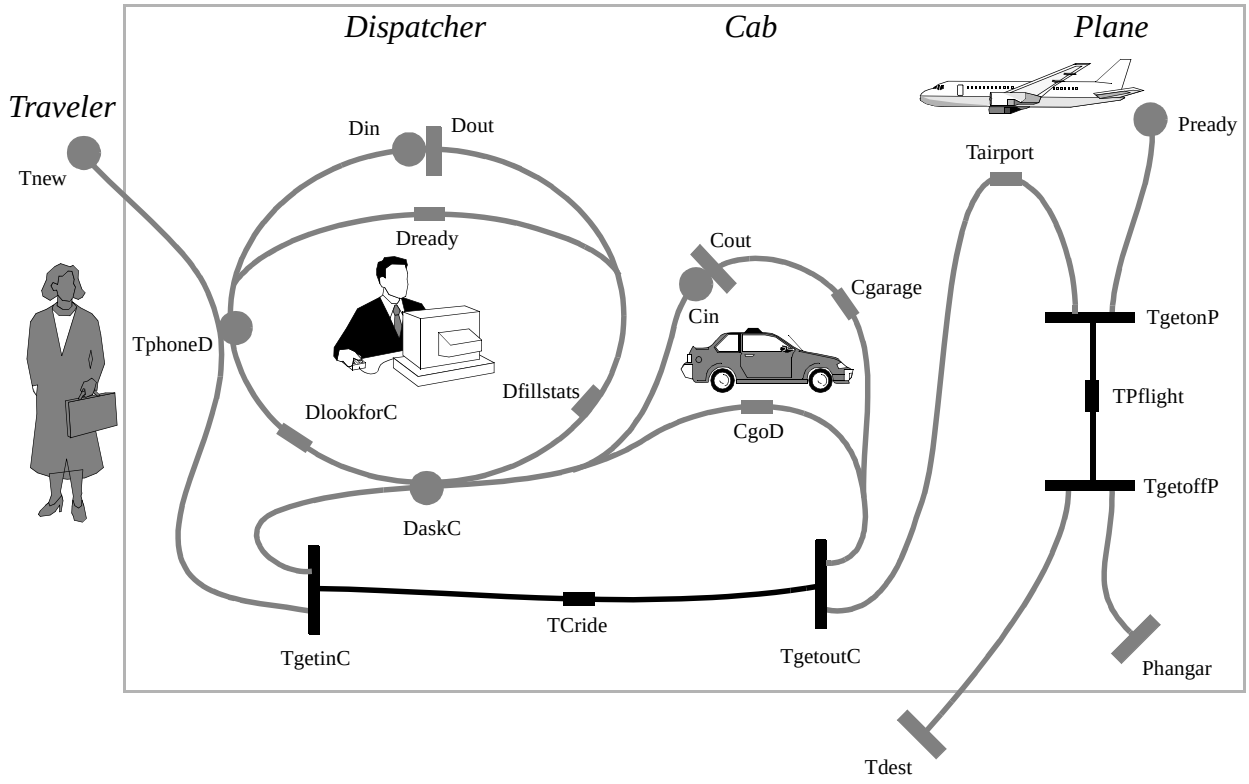
As a more concrete example, we use the *Traveler System* as defined in [Amy 94]. This system is simple to understand and complex enough to test our compiler in a useful way. The purpose of this example is to give a complete view of all files involved in one **tmdl** compilation, and to see that the LOTOS specification effectively works.

6.3.1 The Timethread Map

The map of figure 8 comes from the thesis. It has been slightly altered to satisfy a restriction on loops discussed in section 5.5.5. The timethread *Dispatcher* includes an asynchronous event (`DaskC`) in its loop. Because our prototype compiler cannot manage constructs in loop that lead to parallelism in LOTOS, we transform this asynchronous interaction into a synchronous one. This of course modify the functionality of this system (a dispatcher and a cab now have to synchronize with each other), but this is the best we can do with the current version of the compiler.

Note that the original and grouped LARGs are not given here. Readers can find them in [Amy 94].

FIGURE 8. Timethread map of the new traveler system



6.3.2 TMDL Description

The *STDL* descriptions of the *Traveler System* timethreads are the ones from the thesis and are reproduced in figure 9. To complete the *TMDL* description (file *newtrav.tdl*), we add the internal and interaction sections derived from the grouped LARG.

Timethread descriptions also have their level of specification defined. In our case, they are all set to simple instance with end recursion (`EndRec`).

FIGURE 9. TMDL description of the traveler system (*newtrav.tdl*)

```
{ This is a new version of the traveler example developed in the thesis. }
{ It includes modifications to satisfy compiler's restrictions on loops }
{ Daniel Amyot, 22/09/94 }

Map TaxiCompany is

Internal TPhoneD, TgetinC, TCride, TgetoutC, TgetonP, TPFlight,
 TgetoffP, Pready, Phangar, Din, DaskC, Dout, Cin, Cout

{ Grouping from the thesis }
Interactions
   Traveler, &Gr1 on TphoneD, TgetinC, TCride, TgetoutC,
     TgetonP, TPflight, TgetoffP;
   where
      Group &Gr1 is
         Plane, &Gr2 on Nothing;
         where
            Group &Gr2 is
               Dispatcher, Cab on DaskC;
            EndGroup { &Gr2 }
      EndGroup { &Gr1}

Descriptions

Timethread Traveler is
   EndRec              { End Recursion }
   Internal Tairport   { Internal action }

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

```
Timethread Dispatcher is
   EndRec              { End Recursion }
   Constrained         { Not used here. }
   Internal            { Internal actions }
      DlookforC, Dfillstats, Dready

   Trigger (Din)
   Loop
      Compulsory
         Sync (TphoneD)
         Action (DlookforC)
         Sync (DaskC)  { Instead of Async (Loop restriction). }
         Action (Dfillstats)
      Optional
         Action (Dready)
   EndLoop
   Result (Dout)
EndTT { Dispatcher }

Timethread Cab is
   EndRec              { End Recursion }
   Constrained         { Not used here. }
   Internal            { Internal actions }
      CgoD, Cgarage

   Trigger (Cin)
   Loop                { No parallel here }
      Compulsory
         Sync (DaskC)
         Sync (TgetinC)
         Action (TCride)
         Sync (TgetoutC)
      Optional
         Action (CgoD)
   EndLoop
   Action (Cgarage)
   Result (Cout)
EndTT { Cab }

Timethread Plane is
   EndRec              { End Recursion }
   Trigger (Pready)
   Sync (TgetonP)
   Action (TPflight)
   Sync (TgetoffP)
   Result (Phangar)
EndTT { Plane }

EndMap { TaxiCompany }
```

6.3.3 LOTOS Specification

To get the LOTOS specification from the *TMDL* description, we use the command:

```
tmdl -i newtrav.tdl -o newtrav.lot -t newtrav.tab -e newtrav.err
```

Figure 10 presents the specification (file *newtrav.lot*). As shown by the error file (*newtrav.err*), no errors were found during compilation:

```
TMDL-to-LOTOS Compiler, version 0.9.

Found 0 Warning and 0 Error.

LOTOS specification output.
```

The main points to observe in figure 10 are:

- The *Tag* ADT description (`dummy_val` only since no tags were used),
- The map structure,

- The gate parameters,
- The two loop processes (in *Dispatcher* and *Cab*),
- The management of unique identifers and tabulators.

FIGURE 10. LOTOS specification of the traveler system (*newtrav.lot*)

```
(* TMDL-to-LOTOS Compiler, version 0.9. *)

specification TaxiCompany[Tdest, Tnew]:noexit

library
   Boolean, NaturalNumber
endlib

(* Tag ADT definition *)
type Tag is Boolean, NaturalNumber
sorts Tag
opns dummy_val : -> Tag
     N : Tag -> Nat
     _eq_, _ne_ : Tag, Tag ->Bool
eqns forall x, y: Tag
     ofsort Nat
     N(dummy_val)= 0;   (* dummy value *)
     ofsort Bool
     x eq y = N(x) eq N(y);
     x ne y = not(x eq y);
endtype

behaviour

hide Cin, Cout, DaskC, Din, Dout, Phangar, Pready, TCride, TgetinC,
     TgetoffP, TgetonP, TgetoutC, TPFlight, TPhoneD in

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

where
```

```
process Traveler[TCride, Tdest, TgetinC, TgetoffP, TgetonP, TgetoutC,
                 Tnew, TPflight, TphoneD]:noexit :=

hide Tairport in

   Tnew;
   (
      (
         TphoneD; stop
         |||
         TgetinC;
         TCride;
         TgetoutC;
         Tairport;
         TgetonP;
         TPflight;
         TgetoffP;
         Tdest; Traveler[TCride, Tdest, TgetinC, TgetoffP, TgetonP,
            TgetoutC, Tnew, TPflight, TphoneD]  (* End recursion *)
      )
   )
endproc  (* Timethread Traveler *)

(*****************************************)

process Dispatcher[DaskC, Din, Dout, TphoneD]:noexit :=

hide Dfillstats, DlookforC, Dready in

   Din;
   (
      (
         Loop_0[DaskC, Din, Dout, TphoneD, Dfillstats, DlookforC,
                Dready]>>
           Dout; Dispatcher[DaskC, Din, Dout, TphoneD] (* End recursion *)
      )
   )

where
   process Loop_0[DaskC, Din, Dout, TphoneD, Dfillstats, DlookforC,
                  Dready] : exit :=
      TphoneD;
      DlookforC;
      DaskC;
      Dfillstats;
      (
         (
            Dready;
            Loop_0[DaskC, Din, Dout, TphoneD, Dfillstats, DlookforC,
                   Dready]
         )
         []
         (
                exit  (* Exit Loop *)
         )
      )
   endproc  (* Loop_0 *)

endproc  (* Timethread Dispatcher *)

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

```
process Cab[Cin, Cout, DaskC, TCride, TgetinC, TgetoutC]:noexit :=

hide Cgarage, CgoD in

   Cin;
   (
      (
         Loop_1[Cin, Cout, DaskC, TCride, TgetinC, TgetoutC, Cgarage,
                CgoD]>>
         Cgarage;
         Cout; Cab[Cin, Cout, DaskC, TCride, TgetinC, TgetoutC] (* End
               recursion *)
      )
   )

where
   process Loop_1[Cin, Cout, DaskC, TCride, TgetinC, TgetoutC, Cgarage,
                  CgoD] : exit :=
      DaskC;
      TgetinC;
      TCride;
      TgetoutC;
      (
         (
            CgoD;
          Loop_1[Cin, Cout, DaskC, TCride, TgetinC, TgetoutC, Cgarage,
                 CgoD]
         )
         []
         (
                  exit  (* Exit Loop *)
         )
      )
   endproc  (* Loop_1 *)

endproc  (* Timethread Cab *)

(*****************************************)

process Plane[Phangar, Pready, TgetoffP, TgetonP, TPflight]:noexit :=
   Pready;
   (
      TgetonP;
      TPflight;
      TgetoffP;
      Phangar; Plane[Phangar, Pready, TgetoffP, TgetonP, TPflight]
               (* End recursion *)
   )
endproc  (* Timethread Plane *)

endspec (* Map TaxiCompany *)
```

6.3.4 Symbol Tables

The file *newtrav.tab* contains the symbol tables presented in figure 11. There we can find:

- The map structure.
- The line number associated with the first use of each symbol.
- Global symbol tables: timethreads with defined/used, groups with defined/used, tag values, and activities.
- The map external and internal activities.
- The local symbol tables of the four timethreads: internal activities with type and number of parameters involved, external activities with type and number of parameters involved, and tag identifiers.

FIGURE 11. Symbol tables generated from the traveler system (*newtrav.tab*)

```
TMDL-to-LOTOS Compiler, version 0.9. Symbol Tables.


MAP STRUCTURE:
   TaxiCompany           TYPEGROUP  2 TCride, TgetinC, TgetoffP, TgetonP,
TgetoutC, TPflight, TphoneD
      Traveler              TYPETT     0
      Gr1                   TYPEGROUP  2
         Plane                 TYPETT     0
         Gr2                   TYPEGROUP  2 DaskC
            Dispatcher            TYPETT     0
            Cab                   TYPETT     0
*********************************************************************

TIMETHREAD SYMBOL TABLE:
19:Cab                                  DEFINED      USED
19:Dispatcher                           DEFINED      USED
16:Plane                                DEFINED      USED
12:Traveler                             DEFINED      USED
*********************************************************************

GROUP SYMBOL TABLE:
12:Gr1                                  DEFINED      USED
16:Gr2                                  DEFINED      USED
*********************************************************************

TAG VALUE SYMBOL TABLE:
*********************************************************************

GLOBAL ACTIVITY SYMBOL TABLE:
64:Cgarage
66:CgoD
8:Cin
8:Cout
8:DaskC
45:Dfillstats
8:Din
47:DlookforC
8:Dout
45:Dready
8:Phangar
8:Pready
29:Tairport
7:TCride
38:Tdest
7:TgetinC
8:TgetoffP
7:TgetonP
7:TgetoutC
29:Tnew
7:TPFlight
11:TPhoneD
*********************************************************************
```

```
MAP INTERNALS:
8:Cin
8:Cout
8:DaskC
8:Din
8:Dout
8:Phangar
8:Pready
7:TCride
7:TgetinC
8:TgetoffP
7:TgetonP
7:TgetoutC
7:TPFlight
11:TPhoneD
********************************************************************************

MAP EXTERNALS:
89:Tdest
89:Tnew
********************************************************************************

TIMETHREAD traveler INTERNALS:
29:Tairport                          ACTION       0 PARAMETER.
********************************************************************************

TIMETHREAD traveler EXTERNALS:
32:TCride                            ACTION       0 PARAMETER.
38:Tdest                             RESULT       0 PARAMETER.
31:TgetinC                           SYNC         0 PARAMETER.
37:TgetoffP                          SYNC         0 PARAMETER.
35:TgetonP                           SYNC         0 PARAMETER.
33:TgetoutC                          SYNC         0 PARAMETER.
29:Tnew                              TRIGGER      0 PARAMETER.
36:TPflight                          ACTION       0 PARAMETER.
30:TphoneD                           ASYNC        0 PARAMETER.
********************************************************************************

TIMETHREAD traveler TAG IDENTIFIERS:
********************************************************************************

TIMETHREAD dispatcher INTERNALS:
45:Dfillstats                        ACTION       0 PARAMETER.
47:DlookforC                         ACTION       0 PARAMETER.
45:Dready                            ACTION       0 PARAMETER.
********************************************************************************

TIMETHREAD dispatcher EXTERNALS:
52:DaskC                             SYNC         0 PARAMETER.
47:Din                               TRIGGER      0 PARAMETER.
57:Dout                              RESULT       0 PARAMETER.
50:TphoneD                           SYNC         0 PARAMETER.
********************************************************************************

TIMETHREAD dispatcher TAG IDENTIFIERS:
********************************************************************************

TIMETHREAD cab INTERNALS:
64:Cgarage                           ACTION       0 PARAMETER.
66:CgoD                              ACTION       0 PARAMETER.
********************************************************************************

TIMETHREAD cab EXTERNALS:
66:Cin                               TRIGGER      0 PARAMETER.
77:Cout                              RESULT       0 PARAMETER.
69:DaskC                             SYNC         0 PARAMETER.
71:TCride                            ACTION       0 PARAMETER.
70:TgetinC                           SYNC         0 PARAMETER.
72:TgetoutC                          SYNC         0 PARAMETER.
********************************************************************************

TIMETHREAD cab TAG IDENTIFIERS:
********************************************************************************
```

```
TIMETHREAD plane INTERNALS:
*****************************************************************************

TIMETHREAD plane EXTERNALS:
86:Phangar                              RESULT       0 PARAMETER.
82:Pready                               TRIGGER      0 PARAMETER.
85:TgetoffP                             SYNC         0 PARAMETER.
83:TgetonP                              SYNC         0 PARAMETER.
84:TPflight                             ACTION       0 PARAMETER.
*****************************************************************************

TIMETHREAD plane TAG IDENTIFIERS:
*****************************************************************************
```

6.3.5 Execution of the Specification with LOLA

In order to check the validity of our LOTOS specification, we use the validation tool LOLA on a PC computer.

First of all, LOLA accepted the specification without any syntactic or semantic problem.

Then, we used step-by-step simulation to execute a sequence of action used for the validation of the *Traveler* specification in the thesis. Again, the sequence was accepted and no problem was found (fig. 12).

Finally, using the *TestExpand* command, we tested that it was always possible to reach the resulting event (`tdest`) for all sequences of actions. As shown by figure 13, no undesirable deadlock happened and the test result became a **must pass**.

These simple tests do not prove that our specification is correct, but they show that it is at least possible to play with a generated specification with standard LOTOS tools.

FIGURE 12. Sequence of actions used during step-by-step simulation with LOLA

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

FIGURE 13. Testing the reachability of activity `tdest` with TestExpand command under LOLA

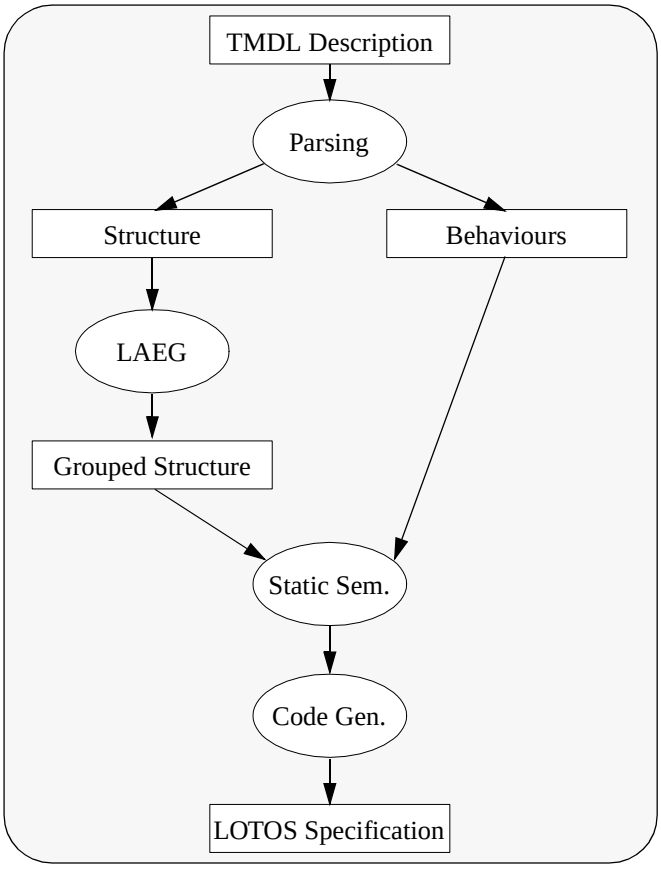
```
LOLA> testexpand 12 tdest -y

    Analysed states       = 1375
    Generated transitions = 2214
    Duplicated states     = 0
    Deadlocks             = 0


    Test result  = MUST PASS.

                   successes = 840
                       stops = 0
                       exits = 0
                cuts by depth = 0
```

7.0 Future Work

Most problems enumerated in this project are related to the fact that we are using a highly-coupled approach in our 1-pass prototype compiler. By redesigning it in a more structured and modular way, it is possible to solve many problems.

Figure 14 presents suggests a new way of implementing the compiler. We believe it makes problems easier to manage, and that it facilitates the modification or addition of functionalities.

FIGURE 14. More structured and complete compiler



Parsing

The parser and scanner rules would not have to be changed in this approach. However, while parsing, the semantic routines would simply generate an internal representation of the TMDL description. The format could be a graph (LARG) representing the structure, and syntax trees representing timethread descriptions (behaviours).

LARG Grouping

From the newly generated LARG, we could get LOTOS-representable groupings by using the LAEG method [Bor 93]. The resulting grouped structure could be a tree like the one used in our prototype compiler.

Static Semantic Analysis

Static semantic routines would be regroups in one module. These routines could validate the structure and behaviour internal trees.

Code Generation

One the trees are validated, code can be generated in an easier way because we can traverse tree nodes to collect all the information we need. We could therefore implement our functions in a much cleaner and clearer way, and find solutions to our current problems related to:

- ADT passing over internal synchronization events (with use of `dummy_val` to get the same number of tag parameters on each gate).
- Implementation and management of `Memory` waiting places and other new ones.
- Implementation the `Loop` construct following the thesis description, therefore eliminating all the loop restrictions.

A separate code generator module would also allow for other LOTOS-like languages to be used as output, without any modification to the parsing, LAEG, and semantic analysis modules.

Of course, this approach would result in a slower 3-pass compiler, but who can't wait a few tenths of a second for a better solution?!

8.0 Conclusion

In this report, we presented an approach to get LOTOS specifications from timethread maps. We use a text representation, called *TMDL*, which is based on BNF rules. Then, a *TMDL* description is compiled into a LOTOS specification that can be used to validate the design.

We presented the language in section 3.0, with its requirements and grammar rules. Then, we showed how this could be used in a compiler. Section 5.0 presented how our TMDL-to-LOTOS prototype compiler was built. We deeply discussed issues and solutions related to static semantic analysis and code generation. Section 6.0 showed some tests applied to the scanner, parser, and code generator. Many complex *TMDL* operators were used. A complete example (the *Traveler System*) was developed, and the resulting specification was rapidly validated. Enhancements to the compiler design are suggested in section 7.0.

We believe this prototype represents another step towards the design of a complex design tool based on timethreads. Current results are very encouraging at this point, and we hope it will raise some interest.

9.0 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)

[Amy 94] D. Amyot, "*Formalization of Timethreads in LOTOS*", M.Sc. Thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada (1994)

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

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

[Bor 93] F. Bordeleau, "*Visual Descriptions, Formalisms and the Design Process*", M.Sc. 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)

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

[FLB 88] C.N. Fischer and R.J. LeBlanc Jr., "*Crafting a Compiler*", The Benjamin/Cummings Publishing Company, Menlo Park (CA), USA (1988)

[GeJ 90] D. Gelernter and S. Jagannathan, "*Programming Linguistics*", MIT Press, Cambridge, USA (1990)

[Jac 93] R. Jacobs, "*Architectural Framework for the Telepresence Voice Server*", Ontario Telepresence Project, Technical Report OTP-93-03 (1993)

[Joh 75] S.C. Johnson, "*Yacc: Yet Another Compiler Compiler*", Computing Science TR-32, Bell Laboratories, Murray Hill (NJ), USA (1975)

[LeS 75] M.E. Lesk and E. Schmidt, "*Lex - A Lexical Analyser Generator*", Computing Science TR-39, Bell Laboratories, Murray Hill (NJ), USA (1975)

[Loc 94] M. Locas "*A Hypergraph-replacement based Notation for Timethread Maps Formal Representation and Transformation*", Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada (1994)

[Man 93] M. Mantei, "*Telepresence User Interface Design Issues and Solutions*", Ontario Telepresence Project, Technical Report OTP-93-02 (1993)

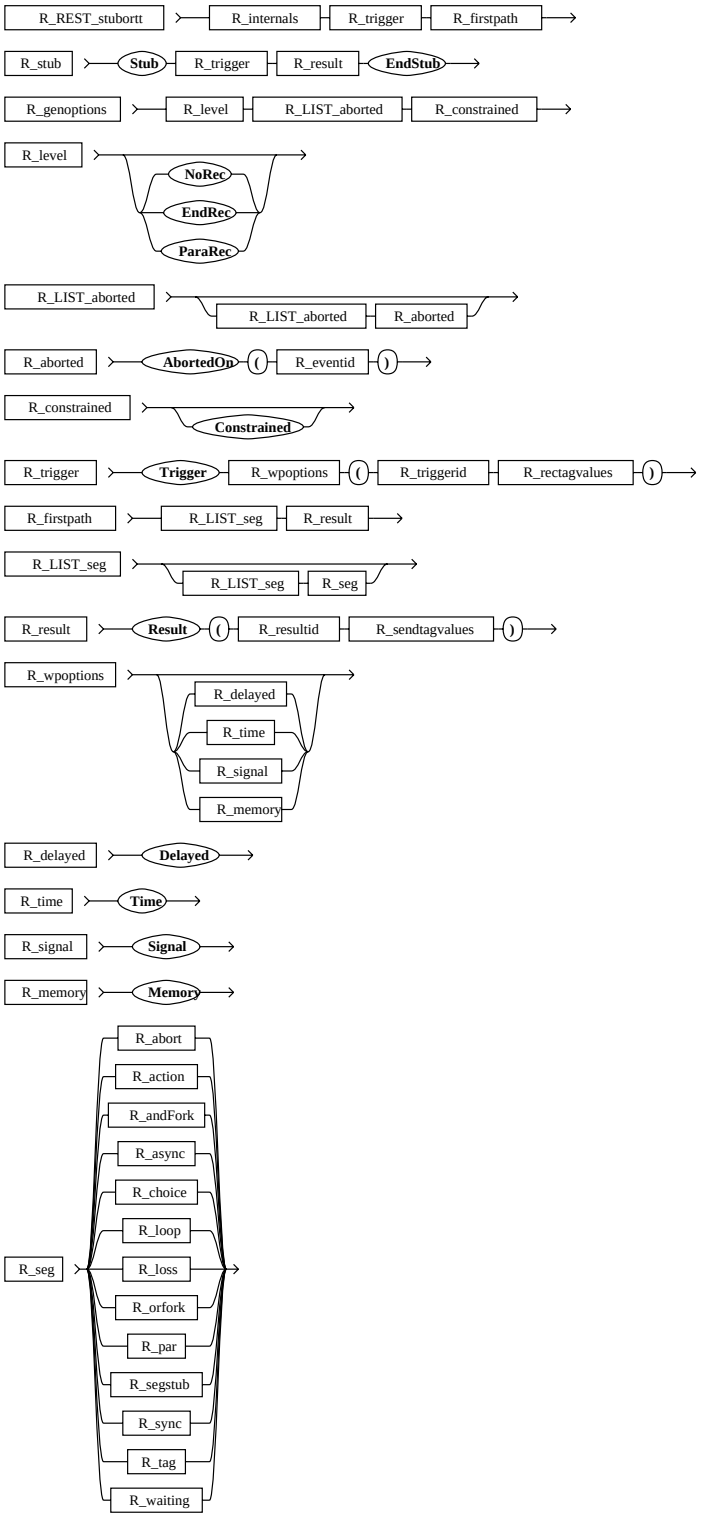
[Mil 92] T. Milligan, "*The telepresence Integrated Interactive Intermedia Facility (iiif)*", Ontario Telepresence Project, Technical Report OTP-93-04 (1992)

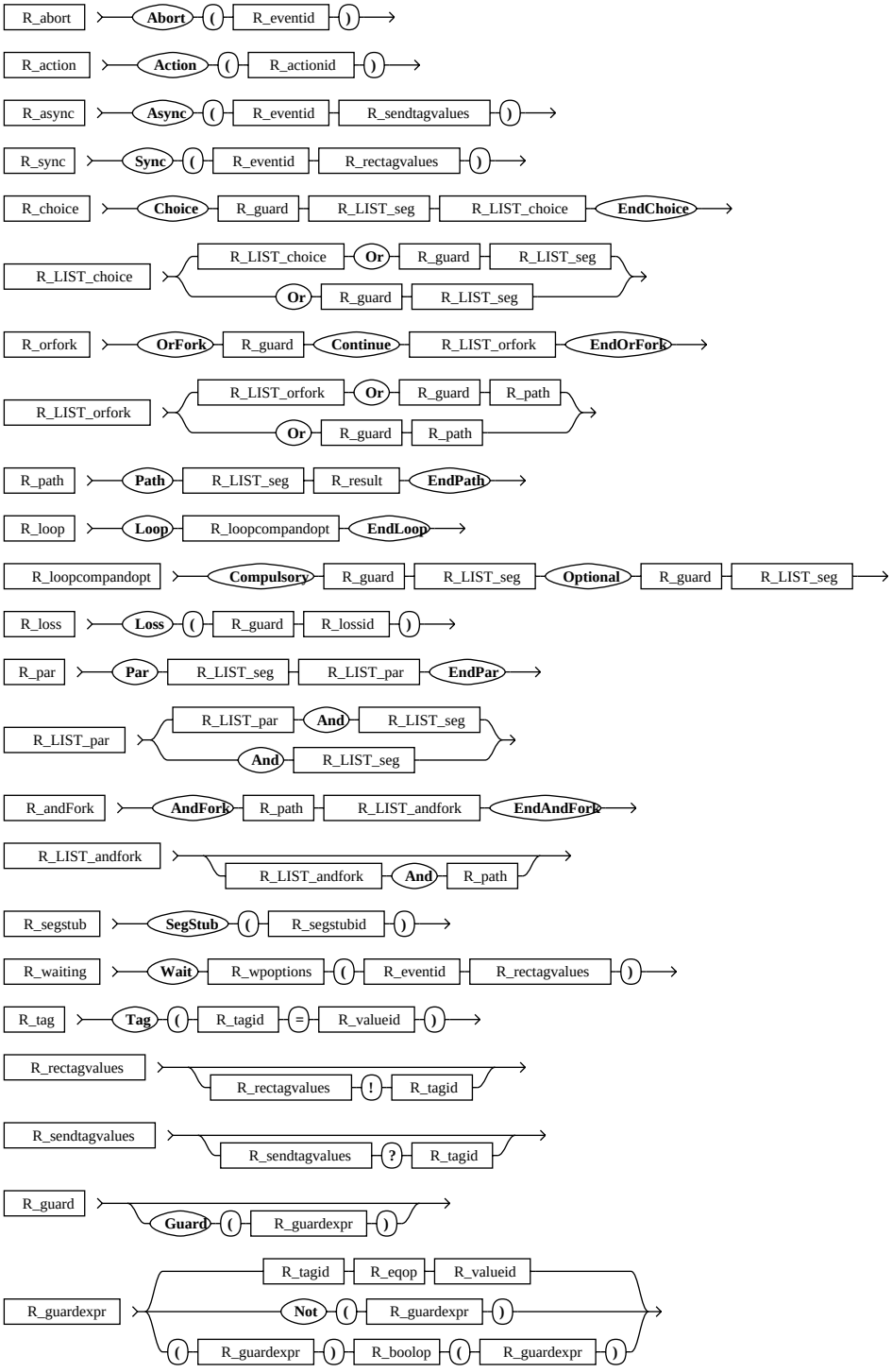
[Val 91] J.J. Valley, "*UNIX Programmer's Reference*", Que Corporation, Carmel (IN), USA (1991)

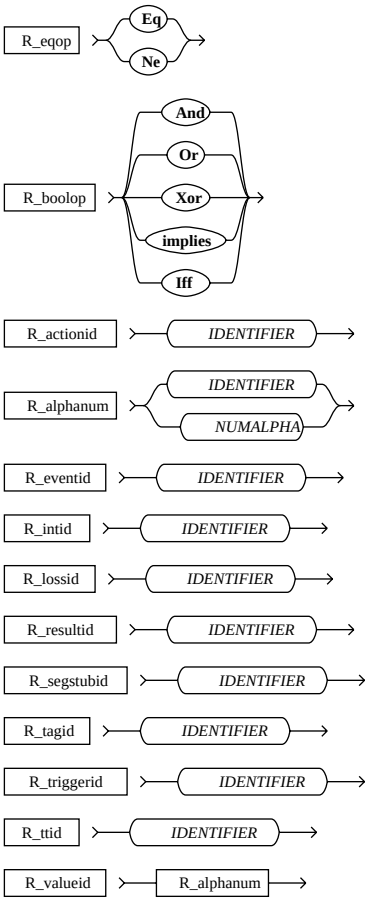
[Wel 93] M. Welsh, "*Linux Installation and Getting Started*", Wilson (NC), USA (1993)

Appendix A TMDL Syntax Diagrams

Here are the BNF syntax diagrams of the *TMDL* grammar rules.


R_REST_stubortt
R_internals
R_trigger
R_firstpath
R_stub
Stub
R_trigger
R_result
EndStub
R_genoptions
R_level
R_LIST_aborted
R_constrained
R_level
NoRec
EndRec
ParaRec
R_LIST_aborted
R_LIST_aborted
R_aborted
R_aborted
AbortedOn
(
R_eventid
)
R_constrained
Constrained
R_trigger
Trigger
R_wpoptions
(
R_triggerid
R_rectagvalues
)
R_firstpath
R_LIST_seg
R_result
R_LIST_seg
R_LIST_seg
R_seg
R_result
Result
(
R_resultid
R_sendtagvalues
)
R_wpoptions
R_delayed
R_time
R_signal
R_memory
R_delayed
Delayed
R_time
Time
R_signal
Signal
R_memory
Memory
R_seg
R_abort
R_action
R_andFork
R_async
R_choice
R_loop
R_loss
R_orfork
R_par
R_segstub
R_sync
R_tag
R_waiting


R_abort
Abort
(
R_eventid
)
R_action
Action
(
R_actionid
)
R_async
Async
(
R_eventid
R_sendtagvalues
)
R_sync
Sync
(
R_eventid
R_rectagvalues
)
R_choice
Choice
R_guard
R_LIST_seg
R_LIST_choice
EndChoice
R_LIST_choice
R_LIST_choice
Or
R_guard
R_LIST_seg
Or
R_guard
R_LIST_seg
R_orfork
OrFork
R_guard
Continue
R_LIST_orfork
EndOrFork
R_LIST_orfork
R_LIST_orfork
Or
R_guard
R_path
Or
R_guard
R_path
R_path
Path
R_LIST_seg
R_result
EndPath
R_loop
Loop
R_loopcompandopt
EndLoop
R_loopcompandopt
Compulsory
R_guard
R_LIST_seg
Optional
R_guard
R_LIST_seg
R_loss
Loss
(
R_guard
R_lossid
)
R_par
Par
R_LIST_seg
R_LIST_par
EndPar
R_LIST_par
R_LIST_par
And
R_LIST_seg
And
R_LIST_seg
R_andFork
AndFork
R_path
R_LIST_andfork
EndAndFork
R_LIST_andfork
R_LIST_andfork
And
R_path
R_segstub
SegStub
(
R_segstubid
)
R_waiting
Wait
R_wpoptions
(
R_eventid
R_rectagvalues
)
R_tag
Tag
(
R_tagid
=
R_valueid
)
R_rectagvalues
R_rectagvalues
!
R_tagid
R_sendtagvalues
R_sendtagvalues
?
R_tagid
R_guard
Guard
(
R_guardexpr
)
R_guardexpr
R_tagid
R_eqop
R_valueid
Not
(
R_guardexpr
)
(
R_guardexpr
)
R_boolop
(
R_guardexpr
)


R_eqop
Eq
Ne
R_boolop
And
Or
Xor
implies
Iff
R_actionid
IDENTIFIER
R_alphanum
IDENTIFIER
NUMALPHA
R_eventid
IDENTIFIER
R_intid
IDENTIFIER
R_lossid
IDENTIFIER
R_resultid
IDENTIFIER
R_segstubid
IDENTIFIER
R_tagid
IDENTIFIER
R_triggerid
IDENTIFIER
R_ttid
IDENTIFIER
R_valueid
R_alphanum

Appendix B Tests

All tests on TMDL constructs are regrouped in one single description. The LOTOS specification follows, and then the symbol tables are given.

B.1 TMDL Description

```
{ This TMDL description tests the complex TMDL constructs. }
{ Daniel Amyot, 20/09/94. }

Map Tests Is

Interactions
   &GrAbort, TestChoice, TestPar, TestForks, TestLoops, TestWP on Nothing;
   Where
      Group &GrAbort B
         TestAbort, &GrAborted on Ab1, Ab2;
         Where
            Group &GrAborted is
              TestAborted1, TestAborted2  on Nothing;
            EndGroup { &GrAborted }
      EndGroup { &GrAbort }

Descriptions

{ These three interacting processes test the Abort and }
{ the three levels of specification. }
Timethread TestAbort Is
   NoRec
   Trigger (T1)
   Abort (Ab1)
   Abort (Ab2)
   Result (R1)
EndTT { TestAbort }

Timethread TestAborted1 Is
   EndRec
   AbortedOn (Ab1)
   Trigger (T2)
   Result (R2)
EndTT { TestAborted1 }

Timethread TestAborted2 Is
   ParaRec
   AbortedOn (Ab2)
   Trigger (T3)
   Result (R3)
EndTT { TestAborted1 }


{ This timethread tests a Choice within a Choice. }
{ The Async activity is also used. }
{ Boolean operators over guards are tested }
{ Finally, tag passing over internal synchronization is tested. }
Timethread TestChoice Is
   Trigger (T4 ? Cond)
   Choice
      Guard (Cond eq Yes)
      Action (Act1)
    OR
      Choice
         { Do nothing }
       OR
         Guard ( (Cond ne Yes) implies (not(Cond eq No)) )
         Async (Act3 ! Cond)
      EndChoice
   EndChoice
   Result (R4)
EndTT { TestChoice }
```

```
{ This timethread tests a Par within a Par. }
{ The Async activity is also used. }
{ Finally, tag passing over internal synchronization is tested. }
Timethread TestPar Is
   Trigger (T5 ? Cond)
   Par
      Wait (Act4)
    AND
      Par
         Async (Act5 ! Cond)
       AND
         Sync (Act6)
      EndPar
   EndPar
   Result (R5)
EndTT { TestPar }


{ Timethread that tests Orfork and AndFork }
{ with three branches. The AndFork is in the OrFork. }
{ Again, tag passing and guards are tested. }
Timethread TestForks Is
   Trigger (T6 ? Cond1 ? Cond2)

   OrFork
      Guard (Cond2 ne No)
      Continue
    OR
      Path
         AndFork
            Path
               Action (Act7)
               Result (Rfork1)
            EndPath
          AND
            Path
               { Do nothing }
               Result (Rfork2 ! Cond1)
            EndPath
          AND
            Path
               Action (Act8)
               Result (Rfork3)
            EndPath
         EndAndFork
         Result (Rfork4)
      EndPath
    OR
      Path
         { Do nothing }
         Result (Rfork5)
      EndPath
   EndOrFork

   Result (R6)
EndTT { TestForks }


{ Tests a loop within a loop, and value passing. }
Timethread TestLoops Is
   ParaRec
   Trigger (T7 ? Cond)
   Loop
      Compulsory
         Guard (Cond ne No)
            Action (Act9)
      Optional
         Guard (Cond eq No)
         Loop
            Comp
               Action (Act10)
            Opt
               Action (Act11)
         EndLoop
```

```
   EndLoop
   Result (R7)
EndTT { TestLoops }


{ Tests the different waiting place options. }
Timethread TestWP Is
   Trigger Time (T8)
   Wait Delayed (Act12)
   Wait Signal (Act13)
   Result (R8 ! TimeT8 ! SigAct13)
EndTT { TestWP }

EndMap { Tests }
```

B.2 LOTOS Specification

```
(* TMDL-to-LOTOS Compiler, version 0.9. *)

specification Tests[Ab1, Ab2, Act1, Act10, Act11, Act12, Act13, Act3,
Act4, Act5, Act6, Act7, Act8, Act9, R1, R2, R3, R4, R5, R6, R7, R8,
Rfork1, Rfork2, Rfork3, Rfork4, Rfork5, T1, T2, T3, T4, T5, T6, T7,
T8]:noexit

library
   Boolean, NaturalNumber
endlib

(* Tag ADT definition *)
type Tag is Boolean, NaturalNumber
sorts Tag
opns dummy_val, No, OK, TOut, Yes : -> Tag
     N : Tag -> Nat
     _eq_, _ne_ : Tag, Tag ->Bool
eqns forall x, y: Tag
     ofsort Nat
     N(dummy_val)= 0;   (* dummy value *)
     N(No) = Succ(N(dummy_val));
     N(OK) = Succ(N(No));
     N(TOut) = Succ(N(OK));
     N(Yes) = Succ(N(TOut));
     ofsort Bool
     x eq y = N(x) eq N(y);
     x ne y = not(x eq y);
endtype

behaviour
   (
      (
         TestAbort[Ab1, Ab2, R1, T1]
         |[Ab1, Ab2]|
         (
            TestAborted1[Ab1, R2, T2]
            |||
            TestAborted2[Ab2, R3, T3]
         )
      )
      |||
      TestChoice[Act1, Act3, R4, T4]
      |||
      TestPar[Act4, Act5, Act6, R5, T5]
      |||
      TestForks[Act7, Act8, R6, Rfork1, Rfork2, Rfork3,Rfork4, Rfork5,T6]
      |||
      TestLoops[Act10, Act11, Act9, R7, T7]
      |||
      TestWP[Act12, Act13, R8, T8]
   )

where
```

```
process TestAbort[Ab1, Ab2, R1, T1]:noexit :=
   T1;
   (
      Ab1;  (* Abort event *)
      Ab2;  (* Abort event *)
      R1; stop  (* No recursion *)
   )
endproc  (* Timethread TestAbort *)

(*****************************************)

process TestAborted1[Ab1, R2, T2]:noexit :=
   (
      T2;
      (
         R2; TestAborted1[Ab1, R2, T2]  (* End recursion *)
      )
   )
   [> ab1; TestAborted1[Ab1, R2, T2]
endproc  (* Timethread TestAborted1 *)

(*****************************************)

process TestAborted2[Ab2, R3, T3]:noexit :=
   (
      T3;
      (
         R3; stop
         |||
         TestAborted2[Ab2, R3, T3]  (* Parallel recursion *)
      )
   )
   [> ab2; TestAborted2[Ab2, R3, T3]
endproc  (* Timethread TestAborted2 *)

(*****************************************)

process TestChoice[Act1, Act3, R4, T4]:noexit :=
hide Sync_Or_0, Sync_Or_1 in

   T4 ? Cond:Tag;
   (
      (
         (
            [Cond eq Yes]->
            Act1;
            Sync_Or_0 ! Cond;stop
         )
         []
         (
            (
               (
                  Sync_Or_1 ! Cond;stop
               )
               []
               (
                  [(Cond ne Yes) implies (not(Cond eq No))]->
                  (
                     Act3 ! Cond; stop
                     |||
                     Sync_Or_1 ! Cond;stop
                  )
               )
            )
            |[Sync_Or_1]|
            Sync_Or_1 ? Cond:Tag;
            Sync_Or_0 ! Cond;stop
         )
      )
      |[Sync_Or_0]|
      Sync_Or_0 ? Cond:Tag;
      R4; stop  (* No recursion *)
   )
endproc  (* Timethread TestChoice *)
```

```
(*****************************************)
process TestPar[Act4, Act5, Act6, R5, T5]:noexit :=
hide Sync_And_2, Sync_And_3 in
   T5 ? Cond:Tag;
   (
      (
         (
            Act4;
            Sync_And_2 ! Cond;stop
         )
         |[Sync_And_2]|
         (
            (
               (
                  (
                     Act5 ! Cond; stop
                     |||
                     Sync_And_3 ! Cond;stop
                  )
               )
               |[Sync_And_3]|
               (
                  Act6;
                  Sync_And_3 ! Cond;stop
               )
            )
            |[Sync_And_3]|
            Sync_And_3 ? Cond:Tag;
            Sync_And_2 ! Cond;stop
         )
      )
      |[Sync_And_2]|
      Sync_And_2 ? Cond:Tag;
      R5; stop  (* No recursion *)
   )
endproc  (* Timethread TestPar *)
(*****************************************)
  process TestForks[Act7, Act8, R6, Rfork1, Rfork2, Rfork3, Rfork4,
Rfork5, T6]:noexit :=
     T6 ? Cond1:Tag ? Cond2:Tag;
     (
        (
           (
              (
                 Act7;
                 Rfork1; stop
                 |||
                 Rfork2 ! Cond1; stop
                 |||
                 Act8;
                 Rfork3; stop
                 |||
                 Rfork4; stop  (* No recursion *)
              )
           )
           []
           (
              Rfork5; stop  (* No recursion *)
           )
           []
           (
           [Cond2 ne No]->
              R6; stop  (* No recursion *)
           )
        )
     )
endproc  (* Timethread TestForks *)
```

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

process TestLoops[Act10, Act11, Act9, R7, T7]:noexit :=
   T7 ? Cond:Tag;
   (
      (
         Loop_4[Act10, Act11, Act9, R7, T7](Cond) >>
            accept Cond :Tag  in
         R7; stop
         |||
        TestLoops[Act10, Act11, Act9, R7, T7]  (* Parallel recursion *)
      )
   )

where
   process Loop_5[Act10, Act11, Act9, R7, T7](Cond :Tag): exit(Tag) :=
      Act10;
      (
         (
            Act11;
            Loop_5[Act10, Act11, Act9, R7, T7](Cond)
         )
         []
         (
                exit(Cond)   (* Exit Loop *)
         )
      )
   endproc  (* Loop_5 *)

   process Loop_4[Act10, Act11, Act9, R7, T7](Cond :Tag): exit(Tag) :=
      Act9;
      (
         (
            [Cond eq No]->
            (
               Loop_5[Act10, Act11, Act9, R7, T7](Cond) >>
                  accept Cond :Tag  in
               Loop_4[Act10, Act11, Act9, R7, T7](Cond)
            )
         )
         []
         (
            [Cond ne No]->
            exit(Cond)   (* Exit Loop *)
         )
      )
   endproc  (* Loop_4 *)

endproc  (* Timethread TestLoops *)

(*****************************************)

process TestWP[Act12, Act13, R8, T8]:noexit :=

hide Delay_8, Sync_Signal_9, Sync_Time_7, TimeOut_6 in

   (
      TimeOut_6;  (* Timeout occured *)
      Sync_Time_7 ! TOut; stop
      []
      T8;  (* Event occured *)
      Sync_Time_7 ! OK; stop
   )
   |[Sync_Time_7]|
   Sync_Time_7 ? TimeT8:Tag;
   (
      Delay_8;  (* Internal delay *)
      Act12;
      (
        Act13; Sync_Signal_9 ! TimeT8 ! Yes; stop  (* Signal occured *)
         []
         Sync_Signal_9 ! TimeT8 ! No; stop  (* No signal occured *)
      )
      |[Sync_Signal_9]|
```

```
         Sync_Signal_9 ? TimeT8:Tag ? SigAct13:Tag;
         R8 ! TimeT8 ! SigAct13; stop  (* No recursion *)
      )
   endproc  (* Timethread TestWP *)

endspec (* Map Tests *)
```

B.3 Symbol Tables

```
TMDL-to-LOTOS Compiler, version 0.9. Symbol Tables.


MAP STRUCTURE:
   Tests                TYPEGROUP  6
      GrAbort              TYPEGROUP  2 Ab1, Ab2
         TestAbort            TYPETT     0
         GrAborted            TYPEGROUP  2
            TestAborted1         TYPETT     0
            TestAborted2         TYPETT     0
      TestChoice           TYPETT     0
      TestPar              TYPETT     0
      TestForks            TYPETT     0
      TestLoops            TYPETT     0
      TestWP               TYPETT     0
*****************************************************************************

TIMETHREAD SYMBOL TABLE:
10:TestAbort                          DEFINED      USED
13:TestAborted1                       DEFINED      USED
13:TestAborted2                       DEFINED      USED
7:TestChoice                         DEFINED      USED
7:TestForks                          DEFINED      USED
7:TestLoops                          DEFINED      USED
7:TestPar                            DEFINED      USED
7:TestWP                             DEFINED      USED
*****************************************************************************

GROUP SYMBOL TABLE:
7:GrAbort                            DEFINED      USED
10:GrAborted                          DEFINED      USED
*****************************************************************************

TAG VALUE SYMBOL TABLE:
57:No                                 DEFINED      USED
146:OK                                 DEFINED      UNUSED
146:TOut                               DEFINED      UNUSED
51:Yes                               DEFINED      USED
*****************************************************************************

GLOBAL ACTIVITY SYMBOL TABLE:
24:Ab1
25:Ab2
52:Act1
135:Act10
137:Act11
147:Act12
148:Act13
58:Act3
71:Act4
74:Act5
76:Act6
96:Act7
106:Act8
130:Act9
147:Delay_8
26:R1
33:R2
40:R3
61:R4
79:R5
119:R6
140:R7
149:R8
```

```
97:Rfork1
102:Rfork2
107:Rfork3
110:Rfork4
115:Rfork5
70:Sync_And_2
73:Sync_And_3
50:Sync_Or_0
54:Sync_Or_1
148:Sync_Signal_9
146:Sync_Time_7
23:T1
32:T2
39:T3
49:T4
69:T5
87:T6
126:T7
146:T8
146:TimeOut_6
*******************************************************************************

MAP INTERNALS:
*******************************************************************************

MAP EXTERNALS:
152:Ab1
152:Ab2
152:Act1
152:Act10
152:Act11
152:Act12
152:Act13
152:Act3
152:Act4
152:Act5
152:Act6
152:Act7
152:Act8
152:Act9
152:R1
152:R2
152:R3
152:R4
152:R5
152:R6
152:R7
152:R8
152:Rfork1
152:Rfork2
152:Rfork3
152:Rfork4
152:Rfork5
152:T1
152:T2
152:T3
152:T4
152:T5
152:T6
152:T7
152:T8
*******************************************************************************

TIMETHREAD testabort INTERNALS:
*******************************************************************************

TIMETHREAD testabort EXTERNALS:
24:Ab1                                  ABORT        0 PARAMETER.
25:Ab2                                  ABORT        0 PARAMETER.
26:R1                                   RESULT       0 PARAMETER.
23:T1                                   TRIGGER      0 PARAMETER.
*******************************************************************************

TIMETHREAD testabort TAG IDENTIFIERS:
```

```
*******************************************************************************

TIMETHREAD testaborted1 INTERNALS:
*******************************************************************************

TIMETHREAD testaborted1 EXTERNALS:
31:Ab1                                  ABORTED      0 PARAMETER.
33:R2                                   RESULT       0 PARAMETER.
32:T2                                   TRIGGER      0 PARAMETER.
*******************************************************************************

TIMETHREAD testaborted1 TAG IDENTIFIERS:
*******************************************************************************

TIMETHREAD testaborted2 INTERNALS:
*******************************************************************************

TIMETHREAD testaborted2 EXTERNALS:
38:Ab2                                  ABORTED      0 PARAMETER.
40:R3                                   RESULT       0 PARAMETER.
39:T3                                   TRIGGER      0 PARAMETER.
*******************************************************************************

TIMETHREAD testaborted2 TAG IDENTIFIERS:
*******************************************************************************

TIMETHREAD testchoice INTERNALS:
50:Sync_Or_0                            CHOICE       0 PARAMETER.
54:Sync_Or_1                            CHOICE       0 PARAMETER.
*******************************************************************************

TIMETHREAD testchoice EXTERNALS:
52:Act1                                 ACTION       0 PARAMETER.
58:Act3                                 ASYNC        1 PARAMETER.
61:R4                                   RESULT       0 PARAMETER.
49:T4                                   TRIGGER      1 PARAMETER.
*******************************************************************************

TIMETHREAD testchoice TAG IDENTIFIERS:
49:Cond                                 DEFINED      USED
*******************************************************************************

TIMETHREAD testpar INTERNALS:
70:Sync_And_2                           PAR          0 PARAMETER.
73:Sync_And_3                           PAR          0 PARAMETER.
*******************************************************************************

TIMETHREAD testpar EXTERNALS:
71:Act4                                 WAIT         0 PARAMETER.
74:Act5                                 ASYNC        1 PARAMETER.
76:Act6                                 SYNC         0 PARAMETER.
79:R5                                   RESULT       0 PARAMETER.
69:T5                                   TRIGGER      1 PARAMETER.
*******************************************************************************

TIMETHREAD testpar TAG IDENTIFIERS:
69:Cond                                 DEFINED      USED
*******************************************************************************

TIMETHREAD testforks INTERNALS:
*******************************************************************************

TIMETHREAD testforks EXTERNALS:
96:Act7                                 ACTION       0 PARAMETER.
106:Act8                                 ACTION       0 PARAMETER.
119:R6                                   RESULT       0 PARAMETER.
97:Rfork1                               RESULT       0 PARAMETER.
102:Rfork2                               RESULT       1 PARAMETER.
107:Rfork3                               RESULT       0 PARAMETER.
110:Rfork4                               RESULT       0 PARAMETER.
115:Rfork5                               RESULT       0 PARAMETER.
87:T6                                   TRIGGER      2 PARAMETERS.
*******************************************************************************
```

```
TIMETHREAD testforks TAG IDENTIFIERS:
87:Cond1                                DEFINED      USED
87:Cond2                                DEFINED      USED
*****************************************************************************

TIMETHREAD testloops INTERNALS:
*****************************************************************************

TIMETHREAD testloops EXTERNALS:
135:Act10                               ACTION       0 PARAMETER.
137:Act11                               ACTION       0 PARAMETER.
130:Act9                                ACTION       0 PARAMETER.
140:R7                                  RESULT       0 PARAMETER.
126:T7                                  TRIGGER      1 PARAMETER.
*****************************************************************************

TIMETHREAD testloops TAG IDENTIFIERS:
126:Cond                                DEFINED      USED
*****************************************************************************

TIMETHREAD testwp INTERNALS:
147:Delay_8                             DELAY        0 PARAMETER.
148:Sync_Signal_9                       TIME         1 PARAMETER.
146:Sync_Time_7                         TIME         1 PARAMETER.
146:TimeOut_6                           TIME         0 PARAMETER.
*****************************************************************************

TIMETHREAD testwp EXTERNALS:
147:Act12                               WAIT         0 PARAMETER.
148:Act13                               WAIT         0 PARAMETER.
149:R8                                  RESULT       2 PARAMETERS.
146:T8                                  TRIGGER      0 PARAMETER.
*****************************************************************************

TIMETHREAD testwp TAG IDENTIFIERS:
148:SigAct13                            DEFINED      USED
146:TimeT8                              DEFINED      USED
*****************************************************************************
```