Published in: K. Parher, G. Rose, eds., IFIP Transactions C-2: Formal Description Techniques IV, Elsevier Science Publishers B.V. (North-Holland), Amsterdam (1992), 313-327.

# A LOTOS Data Facility Compiler (DAFY)[1]

Eric Lallemand
Research Engineer

Guy Leduc
Research Associate F.N.R.S. [2]

Université de Liège, Institut d'Electricité Montefiore, B28, B-4000 Liège 1, Belgium
Tel: + 32 41 562691 Fax: + 32 41 562989
E-mail: u514401@bliulg11.bitnet or leduc@montefiore.ulg.ac.be

## Abstract

If we take a look at existing LOTOS specifications, we notice that the description of the needed data types is very often huge. This causes the lack of concision of most descriptions of complex systems. We propose to tackle this problem in two steps. First, we define extensions to the LOTOS language allowing short definitions of most of the data types used in practical LOTOS specifications. Second, we propose a tool called "**DAFY**" (**Da**ta **F**acility Compiler) which is able to translate these extensions into standard LOTOS.

## 1. Introduction

LOTOS (**L**anguage **O**f **T**emporal **O**rdering **S**pecification) was developed by FDT (Formal Description Technique) experts during the years 1981-1988. The basic idea behind LOTOS is that systems can be described by defining the temporal relation between events describing the externally observable behaviour of a system.

LOTOS has two relatively independent components:
- The first one deals with the description of the process behaviour of a system, and is based on a modification of CCS (Calculus of Communicating Systems - [Mil 80], [Mil 89]) and CSP (Communicating Sequential Processes - [Hoa 85]).
- The second one deals with the description of data structures and value expression and is based on the abstract data type language ACT ONE which was developed at the Technical University of Berlin ([EhM 85]).

Although it has been developed for OSI (Open Systems Interconnection), LOTOS is a Formal Description Technique which is generally applicable to distributed, concurrent information processing systems. LOTOS was developed with the purpose of allowing the production of system descriptions which are **unambiguous**, **complete**, **consistent**, **precise** and **concise**.

In fact, when we take a look at existing specifications, we notice that the description of the needed data types is often huge. This causes the lack of concision of most descriptions of complex systems. This problem has already been identified by G. Scollo in 1986. In order to solve it, he proposed to extend the language definition ([Sco 86]) with shorthand notations to be able to produce concise data type descriptions. However, most of the extensions proposed

[1] This work was partially supported by the Commission of the European Communities under the ESPRIT II programme in project OSI 95.

[2] Fond National de la Recherche Scientifique: National Fund for Scientific Research (Belgium)

by G. Scollo were not included in the standardized definition of the language. Thus, the problem remains unsolved as system descriptions using the extended language are neither internationally accepted by the scientific community nor tractable by LOTOS related tools.

## 2. The language extensions

The shorthand notations, proposed by G. Scollo in 1986 when LOTOS was not a standard yet, intended to ease the data definition work and to speed it up ([Sco 86]).

After a short description of the proposed facilities, we will evaluate their usefulness and we will look at the feasibility to define additional facilities.

### 2.1. The extensions proposed by G. Scollo

#### 2.1.1. "Constant"

This language extension allows the easy definition of data types whose value domains (sorts) consist only of a finite number of constants whose names are the parameters of the extension invocation [1] .

The operations defined within this type allow the comparison of the constants (equality, inequality). These operations are defined considering that constants of different names are different.

A more complete version of this extension also defines additional operations such that "less than", "less or equal", "greater than", "greater or equal". The results provided by these operations are based on an order relation which will somehow have to be defined by the user of this extension. This will be done in relation with the order of the names of the constants in the parameter field of the extension invocation.

#### 2.1.2. "Map"

This facility allows the easy and explicit description of a function (possibly partial) between the two sorts which are the parameters of the extension invocation.

The operations defined by this type allow:
- the association of a (new) value to a term of the function domain;
- the generation of the value associated to a term of the function domain;
- the generation of the function domain;
- the comparison of two functions, …

#### 2.1.3. "OneOf"

LOTOS allows the use of the same name to refer to different operations (if it is possible to distinguish each occurrence of the same identifier because of either the nature of the operation arguments or the relative position of the operator with respect to its arguments). This feature is called "overloading" and it is not allowed to be used for sort identifiers. So it is not possible to use the same identifier to denote two different sorts.

This extension tries to remedy this situation by allowing the definition of a sort as the "union" of other value domains which are the parameters of the extension invocation.

[1] By "extension invocation", we mean the text to be typed by the specifier who uses the language extension.

The operations associated with this type allow the comparison of two terms of the “global” sort and to know to which “subsort” a term belongs.

### 2.1.4. “Set” and “String”

These facilities represent the well-known concepts used in mathematics or in computer science.

### 2.1.5. “Tuple”

This extension allows the definition of a sort as the cartesian product of a finite number of data domains which are the parameters of the extension invocation.

This type is similar to a “record” in PASCAL or to a “struct” in C.

The operations associated with this type allow the construction of a term (i.e. assign an initial value to all fields of the structure), the extraction of the value of a tuple component, the modification of one or several components of a tuple and at last the test of equality of two tuples.

Note that, as for the “Constant” extension, a more complete version of the “Tuple” extension is also available. This extended version defines and uses an order relation over the tuples.

## 2.2. Other extensions of the language ?

It may seem surprising that other abstract data types like those mentioned in [UhS 90] are neither included in the standard library nor proposed to be defined as language extensions. Are “Queues”, “stacks”, “trees”, … (which are very frequently used in ordinary programming language) useless in the LOTOS context ? It seems that the answer of this question is “yes”. Indeed, the study of the protocol and service specifications of the transport and the session layers of the OSI model ([ISO 9571], [ISO 9572], [ISO 10022] and [ISO 10023]) illustrates that:
- Most data types used in the specifications could be specified by using the language extensions proposed by G. Scollo.
- Data types similar to “stacks” or “trees” are not used.
- Data types similar to “queues” are rather seldom. Moreover, each occurrence generally having specific characteristics, it would only be possible to specify a reduced version of this type which only takes account of the common characteristics of all the “queues” (which can be resumed in the “first in, first out” data access policy). Another reason that led us to reject this type as a possible extension is the impossibility to correctly and completely specify operations representing partial functions in ACT ONE. Indeed, a partial function is a function which is not defined on all the values of its domain. This is the case of the function which tries to extract the value of the first element of a queue as the result is undefined if the queue is empty.
- Other data types used in the specification are few, quite short and too particular to consider the creation of facilities enabling their substitution.

These results also take account of the fact that it is possible to specify some data types using processes ([Got 87], [Led 87], [Led 90]).

## 2.3. The treatment of the extensions

As the use of non-standard features is forbidden in LOTOS and as the extensions proposed by G. Scollo seem very useful, we decided to design a tool which translates an extended LOTOS specification (based on variants of these extensions) in a standard LOTOS specification. This tool will ease the specification task in two significant ways:
- it will permit the reduction of the length of the specifications written by specifiers;

- it will allow an easy definition of most of the needed data types.

Although the "Set" and "String" extensions proposed by Scollo were retained in the standard library, we have chosen to treat them because the extension invocations translated by DAFY will be much easier to use than the library types which have to be correctly actualized and renamed.

Although we only consider the extensions proposed by G. Scollo, the tool will be designed in a modular way in order to allow the incorporation of future extensions.

## 3. Definition of the extension invocation syntax and of the translations produced

### 3.1. The extension invocation syntax

When we defined the syntax to be used to invoke the extensions, we had to find a compromise between two opposite desires:

- On one hand, the desire to use a syntax as close as possible to the one used for abstract data types in LOTOS. The use of the extensions will then seem as natural as possible.
- On the other hand, the need for the translation tool to be able to distinguish the use of an extension from a standard data type definition (this is only possible if the syntax used for the extension invocations is different enough from the one used by standard LOTOS).

Taking the technical problem related to the design of a compiler into account, but considering that we wanted to produce a tool easy to use, we chose the following syntax for the extension invocations:

***type*** *type_name* ***isdafy*** *extension_name*
    *[**actualizedby** type_name_list **using**]*
*[**sortnames** sort_name]*
*[**opnnames** (operation_name **for** operation_name)+]*
***parnames*** *identifier_list*
***endtype***

Parts between "[]" are optional and "(operation_name **for** operation_name)+" means the repetition of at least one "(operation_name **for** operation_name)" group.

Example of the use of this syntax:

***type*** *Constant_3* ***isdafy*** *Constant*
***sortnames*** *Const*
***parnames*** *Const_1, Const_2, Const_3*
***endtype***

This example defines the "Constant_3" type and the "Const" sort whose only three terms are "Const_1", "Const_2" and "Const_3". In this invocation of the "Constant" extension, we chose to impose the produced sort name (which is by default chosen equal to the type name) but we did not ask the compiler to modify the operation names which are produced by default. This modification could have been done as in standard LOTOS using the "**opnnames**" keyword.

The only differences between an actualization in standard LOTOS and an extension invocation are:

- The use of the "**isdafy**" (**is da**ta **f**acilit**y**) keyword which effectively indicates that we will use a language extension rather than a previously defined data type;
- The non-use of the "**for** sort_name" after the "**sortnames**" keyword;

- The use of "**parnames**" followed by an identifier list. This word has been used in order to uniformly treat the parameters of all the extensions (the parameters of the "Constant" extension are operation names whereas the parameters of the other extensions are sort names).

Let us note that only the "**isdafy**" word is considered as a new keyword by the data facility compiler. The "parnames" word and the extension names ("Constant", "Map", "OneOf", ...) can be used as normal LOTOS identifiers (except "Constant+" and "Tuple+" which are used to design the extended version of the "Constant" and "Tuple" extensions; they are not standard LOTOS identifiers).

## 3.2 Translation of the extension invocations

### 3.2.1. The generic data type concept

The standard LOTOS allows the description of *parametrized data types* which can be considered as partial specifications where only some general features of the type are described and "holes" (formal sorts, operations and equations) are left to be filled later with further details. An example of such a parametrized data type is the well-known "Set" defined in the standard library annexed to the LOTOS language definition ([ISO IS 8807]). From this "Set" definition, it is possible to easily **generate** the definition of any set whose operations have the same properties as those specified in the initial type. For this reason, "Set" is considered as a **generic** data type definition.

By using the same LOTOS feature, it is possible to define generic data types by defining the operation properties of a "Map" (its domain and its range are the parameters), a "OneOf" for a given number of subsorts gathered (these sorts are the parameters), a "String" (the sort of the elements contained is the parameter), a "Tuple" of a given number of fields (the sorts of the fields are the parameters). An example of such a generic data type produced by DAFY and representing a "Tuple" of three fields is given between the 22nd and the 92nd line of the annex B.

There is no need to use parametrized data types to define sorts consisting only of a finite number of constants as only the constant names are unknown. Nevertheless, because of the *renaming feature* of the standard LOTOS, it is also possible to specify a **generic** type which defines a sort of a given number of constants whose names are adaptable at will. An example of a generic data type defining three constants is given between the 4th and the 20th line of the annex B.

Taking the preceding remark into account, we decided to produce the translations of the extension invocations in two stages:
- the production of a generic data type completely specifying the concepts used by the facility (possibly taking the number of parameters into account) and,
- the adaptation of the generic data type to the particularities of the facility invocation.

These translations will be produced at two different levels of specification by the data facility compiler:
- the generic data types needed by the complete set of extensions used in the source specification will be produced just before the "**behaviour**" symbol in order to give them global scope (cf. lines 4 to 92 of the annex B);
- the translations instantiating the generic data types will be produced where the extension invocations are detected (these translations are said "local"). (cf. lines 97 to 132 of the annex B).

Let us note that it is impossible to define a single generic data type to describe a "Tuple" of any number of fields because the properties of the operations depend upon this number. For the same reason, it is also impossible to define a single generic data type for a "Constant" or a

“OneOf” of any number of parameters. Thus, we will have to produce as many generic data types as the different numbers of parameters used in these three extensions.

### 4.2.2. Partial functions specification

We will not describe each line of the translations (local or generic) produced for each facility but we will rather describe the two major problems we had to face during the study of the translations and the way we solved them.

The semantics of the LOTOS language makes it impossible to describe partial operations completely and correctly ([EhM 85]). Two of the most common solutions used to overcome this problem are:
- “The *error value* solution”: The use of an error value which will be returned if the function is undefined for the values of its arguments.
- “The *set* solution”: The modification of the definition of the problematic operation such that it does not return a single value but a set of values. Thus, if the operation was originally defined for some arguments and returned “v”, it will now return the singleton containing “v” and, if the operation was not defined, it will now return the empty set.

The use of the first solution very often leads to contradictions in the produced equations. The second solution is not free of problems either. Indeed, the insertion of the result of the operation in a set only delays the problem because it is not possible to specify a total function that could extract the solution possibly contained in the set.

Another solution would consist in simply not defining the behaviour of the operation if the represented function is not defined.

Considering that no solution is perfect, we decided initially to use the “set” solution to specify the properties of the partial operations of the “Map” and the “OneOf” facilities. Later, we extended DAFY very quickly (2 hours’ work) in order to support the “undefined” solution.

### 4.2.3. Term rewriting systems

In order to simulate LOTOS specifications, and in particular to evaluate ACT ONE value expressions, the usual technique consists in translating the ACT ONE equations into a term rewriting systems. Classically, the equations of the data types definitions are interpreted as rewrite rules from left to right by the simulator[3]. This means that if the applying conditions of an equation are satisfied and if a portion of an expression matches the left hand part of the equation, then this portion of expression may be replaced by using the right hand part of this equation. The set of equations of the data type specification (also referred to as the equational theory associated with the specification) are then interpreted as what is commonly called a “term rewriting system”.

However there are basic theoretical limitations to this translation process: some equational theories cannot be translated into terminating and confluent term rewriting systems. Consequently, some valid LOTOS specification (w.r.t. the syntax and static semantics defined in the standard [ISO IS 8807]) are not “simulable”.

A term rewriting system is *terminating* ([Der 85]) iff no infinite derivations are possible (i.e. any sequence of successive applications of rewrite rules must terminate, that is derive a term which is irreducible).

[3] This is the case of the HIPPO V2.1 symbolic simulator of the ESPRIT/SEDOS LOTOS Toolset, but most of the other LOTOS tools apply similar translations.

A term rewriting system is *confluent* ([Der 85]) iff each correct value expression has at most one normal form (i.e. by any sequence of successive applications of rewrite rules, in any order, at most one irreducible term may be found).

In practice, when a specified equational theory makes it impossible to derive an ad-hoc terminating and confluent term rewriting system, it is necessary to modify in a significant way the equational theory itself. For example, the set equational theory can be translated if an order relation is correctly defined for the elements contained in the set. This is due to the fact that, to obtain a terminating and confluent rewrite system, one of the n! representations of a set containing n elements must be given a greater importance, and this can only be done if a correct order relation is defined over the elements of the set.

Taking the preceding remark into account, we decided to produce two translations for each data facility:
- The first one is said "theoretical" and is obtained without taking the limitations of simulator tools into account.
- The second one is said "simulable", and takes these limitations[4] into account and thus specifies variants of these types which generates terminating and confluent rewrite systems.

Being simpler, cleaner and more compact, the first one will be used as reference. Being "simulable" the second one will be used to validate the specification.

An example of a "simulable" translation produced by DAFY is given in the annex B. The only difference between this translation and the "theoretical" one corresponding to the same source specification is that lines 15, 17 and 18 are replaced, in the latter, by a single equation:

"*c1 eq c2 = c2 eq c1;*"

In conjunction with the equations of the 13th, 14th and 16th lines, this equation expresses the same properties of the "eq" operation (constants of different names are different).

## 5. Design of the tool

The structure of the extension invocations makes it possible to use a compiler to translate them.

As the standard LOTOS and the proposed extension invocations satisfy the constraints related to the use of "lex" and "yacc" tools of the UNIX environment, we were able to use them to produce DAFY.

Proceeding this way, we certainly produced the tool faster than if we did it by hand, but above all it allows us to write a quite modular source code. Indeed, it is possible to easily modify the code either to extend the number of extensions treated by the compiler, or to modify the translations produced for an extension (in order to, for example, take account of the limitations specific to other simulators), or even to use another syntax for the extensions (provided that the new syntax satisfy the constraints imposed by "lex" and "yacc").

About 6 months work were needed to produce the code of DAFY which is divided into three major parts:
- the code needed to produce the lexical analyzer (about 2800 lines of "lex" code),
- the code needed to produce the syntactic analyzer (about 2500 lines of "yacc" code),
- the code needed to produce the translations of all the data facilities (about 1400 lines of C).

4 Being designed to be integrated with the SEDOS LOTOS Toolset, we will also have to take account of the fact that HIPPO does not support the standardized definition of the LOTOS language ([ISO IS 8807]). It only supports the draft international standard definition ([ISO DIS 8807]). This implies that some features allowed by the final definition of LOTOS are not supported (the renaming combined with the actualization is an example of such a feature).

This last part is contained in different files which contain all the procedures needed to produce the translation of one language extension.

Let us examine the main characteristics of the lexical analyzer and of the syntactic analyzer. They are the most important parts of the DAFY compiler (cf. Figure 1).

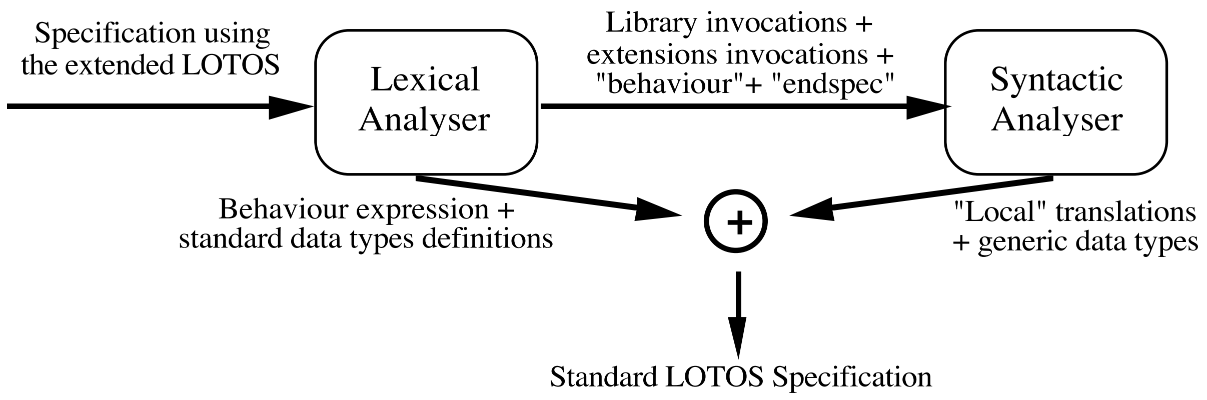
Specification using the extended LOTOS
Lexical Analyser
Library invocations + extensions invocations + "behaviour"+ "endspec"
Syntactic Analyser
Behaviour expression + standard data types definitions
+
"Local" translations + generic data types
Standard LOTOS Specification

**Figure 1: Structure of DAFY.**

## 5.1 The lexical analyzer

The main characteristics of the lexical analyzer of the DAFY program are:

- It correctly identifies the lexical tokens defined in the clause 6.1 of the standard LOTOS definition [5] except the “*)” token which indicates the end of a comment and which must be preceded by a separator (sequence of spaces, of tabulations or carriage return) to be correctly recognized. The “behaviour” symbol is always correctly identified but for reasons of translation, it must be located on a new line (it can only be preceded by separators on its line).
- It correctly identifies the “isdafy” and “parnames” words which are used in the data facilities. These two words are not considered in the same way by the compiler. Indeed, “isdafy” is considered as a keyword (and can not be used as a normal LOTOS identifier) whereas “parnames” is not. This difference results from the fact that “isdafy” actually identifies an extension invocation in comparison to a normal data type definition.
- It recognizes a new type of identifier when it is used in extension invocations. These identifiers are built by using a LOTOS identifier concatenated to the string composed of the “*” character followed by a natural number (let “n” be this natural number). In fact, such an identifier represents a list of “n” LOTOS identifiers separated by a comma.
- It extracts the extension invocations, the library type invocations, the “behaviour” and the “endspec” symbols from the extended LOTOS specification. Only these parts are transmitted to the syntactic analyzer because they are the only parts needed to produce the translations.

[5] This clause defines the basic characters, the keywords, the special symbols and the format of the identifiers used in the standard LOTOS. It also defines the format of the comments which can be used in the specification.

## 5.2. The syntactic analyzer

The main function of the syntactic analyzer (as well as checking the syntax of the extension invocations and reporting the detected errors) is to “decode” the correct extension invocations. The decoding goes with a static semantic check of the following points:

- Is the number of parameters used in the invocation correct ? It must be equal to 1 for the “Set” and “String” extensions and equal to 2 for the “Map” facility;
- Are the renamed operations defined for the referenced facility ?
- Are all the parameters different if they must be so (for the “Constant” and “Map” facilities) ?
- Are all the library types used by the generic translations (which have global scope) invoked before the “behaviour” symbol ? If it is not so, warnings are reported in an error file.

Once the decoding and the semantic check of an extension invocation are finished, the local translation is produced if no error has been reported.

Then, the analyzer indicates in a variable the type of extension used and, if needed, the number of parameters used. This stage is needed in order to produce only the generic data types actually used by the local translations.

If the end of the source specification is reached without any error, the analyzer calls a procedure which inserts the generic data types used by all the local translations just before the “behaviour” symbol. If errors have been detected, they will be reported in an error file whose name must be given when starting DAFY.

Although the analyzer is not always able to correctly identify the nature of the errors encountered during the compilation, their location is correctly reported. This allows an easy debugging of specifications using the extended language. The DAFY compiler recovers rapidly and correctly after the detection of most of the errors.

## 5.3. The resulting translation tool

As the DAFY compiler supports the complete standard LOTOS, it can translate the extension invocations inside a specification using almost any standard LOTOS features. Indeed, in order to produce a specification which is tractable by DAFY, the specifier only needs to check that:

- the “*)” symbol indicating the end of a comment is preceded by at least a separator (space, tabulation or carriage return);
- the “behaviour” symbol is only preceded by separators on its line;
- the “isdafy” word is only used to indicate the use of a language extension (it may not be used as an identifier).

## 5.4. Integration of DAFY in ILOT

ILOT (**I**ntegrated **LOT**OS **T**oolset - [Sch 90]) is a program which was designed to integrate an editor (EMACS) and the tools of the SEDOS LOTOS Toolset (the “SCLOTOS” syntactic analyzer, the “LISA” semantic checker, the “HIPPO” symbolic simulator, …) into a user-friendly environment. It was conceived on a SUN 3 workstation and uses many of the resources provided by SunView (Sun Visual Integrated Environment for Workstations) to ease the design and the treatment of a LOTOS specification.

In order to treat a specification using the extended language as easily as those using the standard language only, we have integrated DAFY in ILOT (cf. figure 2). With this new version of ILOT, it is almost possible to treat a specification without knowing that extensions of the language are used.

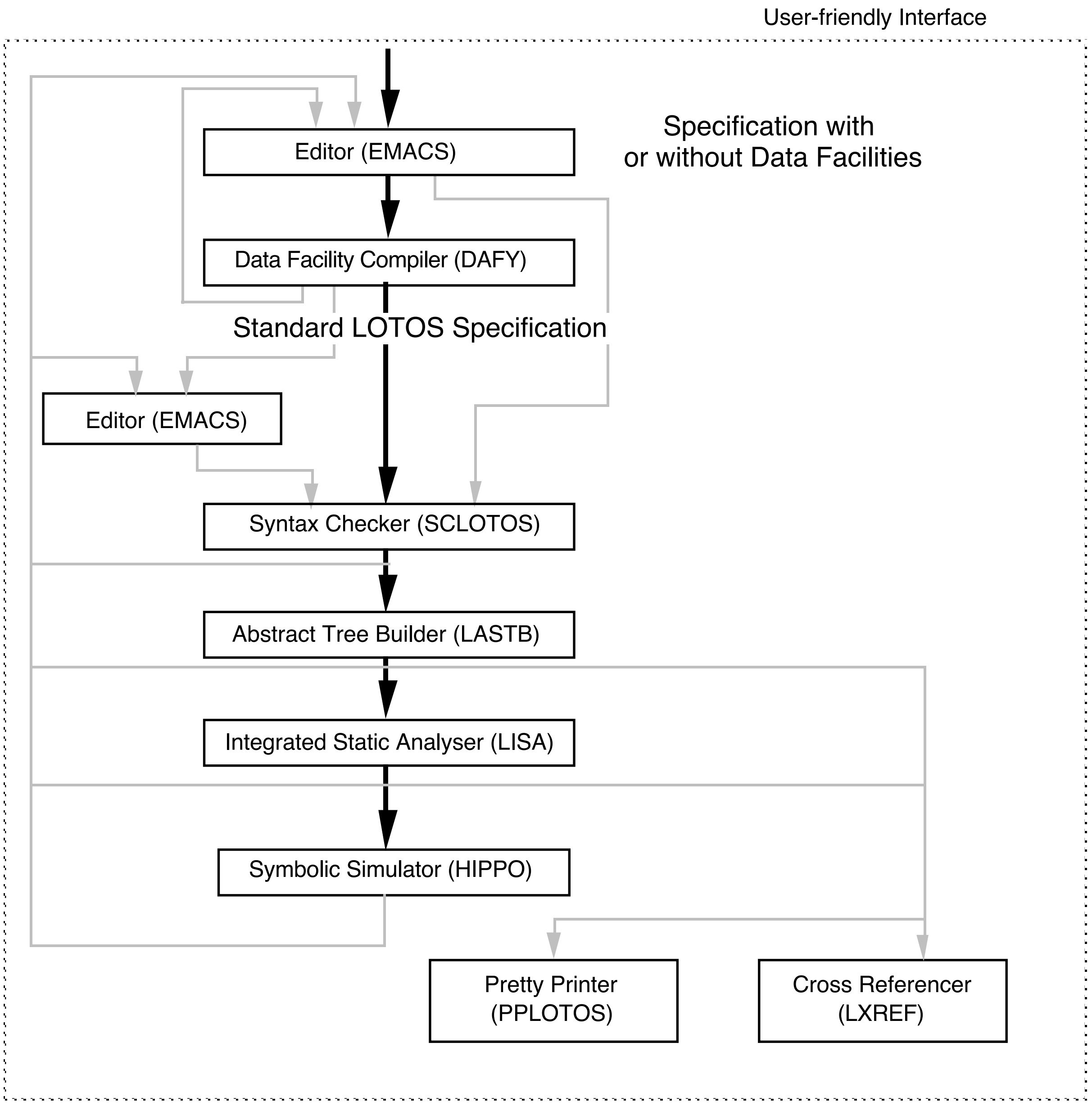
User-friendly Interface
Specification with or without Data Facilities
Editor (EMACS)
Data Facility Compiler (DAFY)
Standard LOTOS Specification
Editor (EMACS)
Syntax Checker (SCLOTOS)
Abstract Tree Builder (LASTB)
Integrated Static Analyser (LISA)
Symbolic Simulator (HIPPO)
Pretty Printer (PPLOTOS)
Cross Referencer (LXREF)

**Fig. 2: Structure of ILOT.**

## 6. Conclusion

We have developed a tool (called "**DAFY**" **- Da**ta **F**acilit**y** Compiler) which translates language extensions into standard LOTOS. It allows the concise description of most of the data types frequently used in LOTOS specifications. We have integrated this tool into the user-friendly "ILOT" environment running on a SUN workstation.

The use of DAFY for the specification of the Transport Protocol showed us that this program allows the reduction of the size of the data type descriptions to about 60 % of the length of those using only standard language data types. The gain obtained by students with small examples has even been greater as they needed less than 30 lines of extension invocations to define all the necessary data types, while the translation of the invocations produced more than 300 lines. DAFY has been defined to produce a quite optimal code. Nevertheless, it is still possible to shorten the size of the produced code by suppressing the lines corresponding to operations which are automatically produced in the translation but not used in the remaining part of the specification.

When designing the tool, we wanted it to be:
- Easy to use. This led us to choose a syntax very close to the one used in standard LOTOS data type definitions.
- Modular in order to be able to extend it easily. Some examples of extensions that we could imagine are: the treatment of newly added language extensions (if the syntax used for the new extension is similar to the one used at the moment), to allow the production of other translations for the partial operations, to take the particularities and limitations of other LOTOS simulators into account, …
- Portable : DAFY, initially developed on a “SUN 3” station, has been easily ported on a “SUN/SPARC” station and on a “DEC 3100” workstation.

## 7. Acknowledgments

We would like to thank professor A. Danthine of the “Systèmes et Automatique” department of the University of Liège who proposed the design of a data facility compiler. We would also like to thank F. Marso and Ch. Pecheur who helped us to realize DAFY. We are grateful to France Bierbaum, Professor Danthine’s assistant, and to the students attending the course “Protocole de Réseaux d’ordinateurs”. They have been the first users of this program and they provided valuable feedback to improve it.

## 9. Bibliography

[Der 85] N. Dershowitz,
*Termination,*
in: J.-P. Jouannaud, ed., Rewriting Techniques and Applications, LNCS 202 (Springer-Verlag, Berlin Heidelberg New York Tokyo, 1985, ISBN 3-540-15976-2) 180-224.

[EhM 85] H. Ehrig and B. Mahr,
*Fundamentals of Algebraic Specification 1, Equations and Initial Semantics,*
in: W. Brauer, B. Rozenberg, A. Salomaa, eds., EATCS , Monographs on Theoretical Computer Science (Springer Verlag, Berlin Heidelberg New York Tokyo, 1985, ISBN 3-540-13718-1).

[Got 87] R. Gotzhein,
*Specifying abstract data types with LOTOS,*
in: G.v. Bochmann, B. Sarikaya, eds., Protocol Specification, Testing and Verification, VI (North-Holland, Amsterdam, 1987, ISBN 0-444-70126-5) 15-26.

[Hoa 85] C.A.R. Hoare,
*Communicating Sequential Processes,*
(Prentice-Hall International, London, 1985, ISBN 0-13-153271-5).

[ISO DIS 8807] ISO/TC97/SC21/WG1/FDT/C,
*LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*,
DIS 8807, July 1987.

[ISO IS 8807] ISO/IEC-JTC1/SC21/WG1/FDT/C,
*Information Processing Systems - Open Systems Interconnection - LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour,*
IS 8807, February 1989.

[ISO 9571] ISO/IEC-JTC1/SC21/WG6 Ad-hoc group,
*Information Technology - Open Systems Interconnection - LOTOS Description of the Session Service*,
TR 9571, Sept. 1989.

[ISO 9572] ISO/IEC-JTC1/SC21/WG6 Ad-hoc group,
*Information Technology - Open Systems Interconnection - LOTOS Description of the Session Protocol,*
TR 9572, Sept. 1989.

[ISO 10023] ISO/IEC-JTC1/SC6/WG4 Ad-hoc Group,
*Formal Description of ISO 8072 in LOTOS,*
DTR 10023, ISO/IEC-JTC1 N1519, Aug. 1991.

[ISO 10024] ISO/IEC-JTC1/SC6/WG4 Ad-hoc Group,
*Formal Description of ISO 8073 in LOTOS,*
Revised text for DTR 10024, ISO/IEC-JTC1/SC6 N6978, Aug. 1991.

[Lal 90] E. Lallemand
*Développement d'un outil d'aide à la définition des types de données en LOTOS*,
Graduate dissertation, University of Liège, June 1990.

[Led 87] G. Leduc,
*The Intertwining of Data Types and Processes in LOTOS,*
in: H. Rudin, C.H. West, eds., Protocol Specification, Testing and Verification, VII, (North-Holland, Amsterdam, 1987, ISBN 0-444-70293-8) 123-136.

[Led 90] G. Leduc,
*Process-oriented and data-oriented specifications in LOTOS*
Thesis annexed to the Agrégation dissertation, Université de Liège, Dept. Systèmes et Automatique, B28, B-4000 Liège, Belgium, 1990.

[Mil 80] R. Milner,
*A calculus of communicating systems,*
LNCS 92 (Springer-Verlag, Berlin Heidelberg New York, 1980, ISBN 3-540-10235-3).

[Mil 89] R. Milner,
*Communication and Concurrency,*
(Prentice-Hall International, London, 1989, ISBN 0-13-114984-9).

[Sch 90] F. Schumacker,
*Interface conviviale à un ensemble d'outils LOTOS*,
Bulletin scientifique de l'Association des Ingénieurs Electriciens sortis de l'Institut d'Electricité Montefiore, 1/1990.

[Sco 86] G. Scollo,
*Some facilities for concise data type definitions in LOTOS,*
Rept. ESPRIT/SEDOS/C1/WP/13/T, University of Twente, March 1986, also in: Potential Enhancements to LOTOS, ISO/TC97/SC21 N2015.

[UhS 90] J. Uhl, H.A. Schmid,
*A Systematic Catalogue of Reusable Abstract Data Types*,
LNCS 460 (Springer-Verlag, New York Berlin Heidelberg, 1990, ISBN 0-387-53229-3).

## Annex A : Specification using some extensions of the language

```
specification demo : noexit
library Boolean, Element, FBoolean, NaturalNumber, String,
Octet endlib

behaviour
stop
where
type Address isdafy CONSTANT
sortnames Add
parnames Add1, Add2, Add3
endtype

type Data isdafy STRING actualizedby Octet using
parnames Octet
endtype

type Packet isdafy TUPLE actualizedby Address, Data using
opnnames DstAdd for comp1
         SrcAdd for comp2
         Data   for comp3
parnames Add*2, Data
endtype
endspec
```

## Annex B : Translation produced by DAFY

```
specification demo : noexit
library Boolean, Element, FBoolean, NaturalNumber, String,
Octet endlib

type dafy_constant3 is Boolean
sorts dafy_constant3
opns const1 : -> dafy_constant3
     const2 : -> dafy_constant3
     const3 : -> dafy_constant3
     _eq_, _ne_ : dafy_constant3, dafy_constant3 -> bool
eqns forall c1, c2 : dafy_constant3
     ofsort bool
     c1 eq c1 = true;
     const1 eq const2 = false;
     const1 eq const3 = false;
     const2 eq const1 = false;
     const2 eq const3 = false;
     const3 eq const1 = false;
     const3 eq const2 = false;
     c1 ne c2 = not (c1 eq c2);
endtype

type dafy_tuple_el1 is Element renamedby
sortnames dafy_tuple_el1 for Element
endtype

type dafy_tuple_el2 is Element renamedby
```

```
sortnames dafy_tuple_el2 for Element
endtype

type dafy_tuple_el3 is Element renamedby
sortnames dafy_tuple_el3 for Element
endtype

type dafy_tuple3_basic is
   dafy_tuple_el1, dafy_tuple_el2, dafy_tuple_el3
sorts dafy_tuple3
opns cons : dafy_tuple_el1, dafy_tuple_el2, dafy_tuple_el3
          -> dafy_tuple3
     comp1 : dafy_tuple3 -> dafy_tuple_el1
     comp2 : dafy_tuple3 -> dafy_tuple_el2
     comp3 : dafy_tuple3 -> dafy_tuple_el3
eqns forall el1 : dafy_tuple_el1,
            el2 : dafy_tuple_el2,
            el3 : dafy_tuple_el3
     ofsort dafy_tuple_el1
         comp1 (cons (el1,el2,el3)) = el1;
     ofsort dafy_tuple_el2
         comp2 (cons (el1,el2,el3)) = el2;
     ofsort dafy_tuple_el3
         comp3 (cons (el1,el2,el3)) = el3;
endtype

type dafy_tuple3_new is dafy_tuple3_basic
opns new_comp1 : dafy_tuple3, dafy_tuple_el1 -> dafy_tuple3
     new_comp2 : dafy_tuple3, dafy_tuple_el2 -> dafy_tuple3
     new_comp3 : dafy_tuple3, dafy_tuple_el3 -> dafy_tuple3
eqns forall t1 : dafy_tuple3,
            el1 : dafy_tuple_el1,
            el2 : dafy_tuple_el2,
            el3 : dafy_tuple_el3
     ofsort dafy_tuple3
        new_comp1 (t1, el1) = cons (el1, comp2(t1), comp3(t1));
        new_comp2 (t1, el2) = cons (comp1(t1), el2, comp3(t1));
        new_comp3 (t1, el3) = cons (comp1(t1), comp2(t1), el3);
endtype

type dafy_tuple3_expanded is dafy_tuple3_new
opns
 new_comp1_comp2 : dafy_tuple3, dafy_tuple_el1, dafy_tuple_el2
                 -> dafy_tuple3
 new_comp1_comp3 : dafy_tuple3, dafy_tuple_el1, dafy_tuple_el3
                 -> dafy_tuple3
 new_comp2_comp3 : dafy_tuple3, dafy_tuple_el2, dafy_tuple_el3
                 -> dafy_tuple3
 new_comp1_comp2_comp3 : dafy_tuple3, dafy_tuple_el1,
                         dafy_tuple_el2, dafy_tuple_el3
                       -> dafy_tuple3
eqns forall t1 : dafy_tuple3,
            el1 : dafy_tuple_el1,
            el2 : dafy_tuple_el2,
            el3 : dafy_tuple_el3
     ofsort dafy_tuple3
 new_comp1_comp2 (t1, el1, el2) = cons (el1, el2, comp3(t1));
 new_comp1_comp3 (t1, el1, el3) = cons (el1, comp2(t1), el3);
 new_comp2_comp3 (t1, el2, el3) = cons (comp1(t1), el2, el3);
 new_comp1_comp2_comp3 (t1, el1, el2, el3) = cons (el1, el2,
                                                         el3);
endtype

type dafy_tuple3 is dafy_tuple3_expanded
opns _eq_, _ne_ : dafy_tuple3, dafy_tuple3 -> fbool
eqns forall t1, t2 : dafy_tuple3
     ofsort fbool
        t1 eq t2 =
            (((comp1 (t1) eq comp1 (t2)) and
            (comp2 (t1) eq comp2 (t2))) and
            (comp3 (t1) eq comp3 (t2)));
        t1 ne t2 = not (t1 eq t2);
endtype

behaviour
stop
where
type Address is dafy_constant3 renamedby
sortnames Add for dafy_constant3
opnnames
          Add1 for const1
          Add2 for const2
          Add3 for const3
endtype


type Data_basic is String renamedby
sortnames Data for string
endtype

type Data is Data_basic actualizedby
   Octet, Boolean, NaturalNumber using
sortnames Bool for FBool
          Nat for Fnat
          Octet for element
endtype


type Packet_basic is dafy_tuple3 renamedby
sortnames Packet for dafy_tuple3
opnnames
          DstAdd for comp1
          SrcAdd for comp2
          Data for comp3
endtype

type Packet is Packet_basic actualizedby
   Address, Data, Boolean using
sortnames Bool for FBool
          Add for dafy_tuple_el1
          Add for dafy_tuple_el2
          Data for dafy_tuple_el3
endtype

endspec
```