VLib : infinite virtual libraries for LOTOS

Charles PECHEUR*

University of Liège, Institut Montefiore, Bât. B28, B-4000 Liège, Belgium
E-Mail: pecheur@montefiore.ulg.ac.be

Abstract

This paper introduces VLib, an extension to LOTOS that eases the specification of data types. It allows the definition and use of virtual libraries containing conceptually an infinite number of data types. The extraction of a working finite subset for a particular application is also considered. After a general introduction to LOTOS data specification issues, the extension is explained and formally defined. A prototype tool implementing the extraction operation is described, and a specification discipline is proposed that benefits from the provided gain of functionality.

Keyword codes:D.3.3; D.3.4
Keywords: Programming Languages, Language Constructs and Features; Processors

1. INTRODUCTION

The formal specification language LOTOS was standardized by ISO [ISO89] and has been applied to numerous developments, both in the applicative and theoretical domains. LOTOS combines two clearly distinguished concepts: process algebras for describing behaviours and algebraic specifications for defining data structures. Both are oriented towards a high abstraction level supported by a strong mathematical basis. The latter is based on the language ACT ONE [Ehrig85] and will be at the centre of the study presented here.

ACT ONE is a powerful tool for reasoning about the properties of data structures. With use however it appears that it is not so well fitted for its practical use in the specification of real systems: whereas its considerable expression power is generally under-used, the very basic nature of its building elements (sorts, operations and equations) and the rigidity of its model lead to lengthy specifications with lots of repetitive, technical details that clutter the data part of the specification. This has been universally recognized as one of the major obstacles against a widespread use of LOTOS by distributed systems designers, and will surely be a major concern for a future revision of LOTOS by ISO [Brinksma92].

In the present situation however, the only recognized language is the one defined in the standard. Moreover all supporting software tools are based on that standard: since one can hardly afford any specification effort without such tools, usage of standard ACT ONE remains the rule.

We propose in this paper a slight extension of the concept of library that is available in standard LOTOS. We introduce *virtual libraries* containing conceptually an infinite number of data type definitions. This allows us to cover collections of common types such as tuples, which could not appear in the standard library because they are infinite. To define virtual libraries, generic macro constructions are added to the language. According to the types

---

* Research Assistant for the National Fund for Scientific Research of Belgium.

requested in a given specification, these constructions are then expanded to generate a finite *real library* in standard ACT ONE.

We assume that the reader is already familiar with ACT ONE. Annex E of the LOTOS standard [ISO89] gives a more detailed presentation and [Ehrig85] contains a much deeper study of the theory of algebraic specifications. Some of the main features are reviewed along the next section.

Section 2 examines ACT ONE in more detail and presents the arguments that lead to our proposal. Section 3 describes the several components of the language extensions that are proposed. Section 4 describes a prototype software tool performing the real library extraction. Finally we discuss some possible variants in section 5, propose a data specification discipline in section 6 and draw some overall conclusions in section 7.

2. IMPROVING ACT ONE

In this section we will examine ACT ONE, both as a theoretical description technique and as a software-supported data definition tool. we will enumerate some of its weaknesses and discuss enhancements which have been proposed to overcome them. We will finally present the arguments that led to the original approach presented in this paper.

2.1. ACT ONE issues

The weaknesses of the data definition part of LOTOS have already been the target of many discussions. They can be resumed to the following categories:

1. *abstraction level*: ACT ONE defines data types in terms of data domains called sorts, operations over these sorts and equations generating equivalence classes within the sorts. These are very basic building blocks, that offer considerable flexibility in the type definitions but lead to very lengthy specifications with lots of trivial statements.
2. *conciseness*: the syntax of the language offers only all-purpose constructs. There is no ad-hoc syntax for frequently used structures (records, sets…) or expressions (numbers, mix fix operators…), which could reduce the specification size and improve its readability. Moreover the syntactic style itself is rather wordy.
3. *higher order constructs*: in order to reduce the specification size and improve its readability, it is desirable to be able to encompass common properties of a family of similar types (sets, records…) under a common notation. ACT ONE provides parameterized types, but a more general solution is needed for some common families.
4. *rigid model*: the model used by ACT ONE imposes other restrictions: all operations are total, and all sorts are disjoint. If partial functions or order-sorted types are needed, they must be simulated explicitly at the specification level. For example, sorts with common elements will be modelled as disjoint sorts with explicit conversion functions.

All these aspects result in long, repetitive and rapidly obscure specifications. Much space is spent describing obvious properties of data types such as equality predicates, which are usually intrinsic in more pragmatic models.

ACT ONE offers some tools for limiting repetitive definitions, such as renaming for generating isomorphic types, actualization of parameterized types containing formal sorts and operations and provision of a standard library. However these fail to support some common data structures, most notably the ubiquitous tuple construct[1]. Simple and well-behaved specifications are easily obtained for a tuple with given member sorts, or with formal member parameters for a given number of members. But such parameterized types have a fixed signature, whereas tuples with a different number of members require a different number of formal sorts, different number of arguments to the constructor operation, etc.

[1]Formally, tuples are elements of a cartesian product of domains. They appear under different names in most programming languages, e.g. *records* in Pascal or *structures* in C.

Therefore tuples of different sizes need different type definitions. This size and thus the number of needed definitions are not bounded, and therefore the library cannot include all of them[1].

2.2. ACT ONE with tools

The difficulties exposed in the preceding section depend only on the ACT ONE itself, as a formal language with a mathematical basis. However LOTOS is hardly used in this form in practical situations: the development and analysis of a LOTOS specification by hand quickly becomes untractable when the specification grows. A flourishing collection of software tools provide assistance in such various task as editing, exploring, compiling, testing and verifying LOTOS specifications. The sound mathematical basis of LOTOS has probably helped the development of such tools, but the computational complexity of the LOTOS model, and especially the initial algebra model of ACT ONE, also opposes great challenges to the tool developers and puts even more restrictions upon the specifier.

The main difficulty is a direct consequence of the level of abstraction used: the equality of two terms is only semi-decidable. This means that there is no general algorithm that can determine whether two terms are equal according to a given specification. Therefore no software tool can handle correctly and completely all ACT ONE specifications: instead, they use derived strategies like term rewriting [Alderden89, Garavel89] or narrowing [Eertink92] which are adequate only for certain restricted classes of definitions[2]. Since the specifier needs data computations to evaluate its specification, he is brought to respect those restrictions and to distort its definitions according to the tool that he will use. This sometimes includes semantic changes: for example, a usable set type definition is only possible if an ordering relation on its member sort is available.

Even for evaluable specifications, the general interpretation of ACT ONE specifications is hungry in computation time and space. To optimize performance, other approaches allow the replacement of some sorts and operations by similar constructive data types taken from conventional programming languages like C, as in [Manas88]. This allows efficient implementations but the validity of the substitution is left to the appreciation of the specifier.

Another problem, rooted in the ACT ONE model, appears acutely in simulation tools: the global propagation of specification errors. It is usually desired that a type definition be *conservative*, which means that it does not alter the meaning of other types from which it is derived [Ehrig85]. This property is not enforced by the ACT ONE model: each new equation potentially affects the whole specification. For example, an erroneous equation resulting in *true* = *false* in any type definition destroys the properties of boolean values over the whole specification. Therefore, when an incorrect data evaluation is detected, it is often difficult to localize the equation which generates it.

2.3. Existing enhancements

Several approaches and tools have been proposed for easing data type definitions in LOTOS. The most straightforward solution is to define compact notations for common type families (sets, records…). Such an approach is proposed in [Scollo86] and used in [Lallemand91] and, in a more rudimentary way, by the EXSTEND facility of the Lotosphere LOTOS Toolset [Eijk91]. The meaning of those notations is given by their translation in standard LOTOS. An alternative is to define a translation in LOTOS of another type definition language: [Bochmann89], [Brady90] and [Thomas92] follow this approach in different ways with ASN.1, another language normalized by ISO. Yet another approach

---

[1] Definitions up to an arbitrary size could have been given, but this solution would not be very satisfying and is not followed by the LOTOS standard.

[2] Admittedly, most equations are thought with an orientation in mind. Indeed, it is quite arduous to specify abstract data types reliably without following such kinds of specification discipline.

proposed in [Brinksma92] is to allow for the definition of implicit operators (e.g. constructors, equality and ordering relations) for some types[1].

Because they keep standard LOTOS as their reference model, such approaches help reducing the size of the specification but do not attempt to resolve model-specific problems such as the definition of partial functions or of conservative extensions. However, the use of extended notations may help to avoid such problems, as will be further explained in section 6. Also note that other tools are available that check conservation properties explicitly [Boehm89,Eertink92].

Defining extensions in the denotation or interpretation of type definitions can present several drawbacks:

- Since the compact notations are not standard, they cannot be used in a published specification (unless their formal definition is annexed, which would usually annihilate the size reduction that they offer). They are thus essentially a local development facility.
- If a pre-existing language like ASN.1 is used, discrepancies between its underlying model and that of ACT ONE require technical hacks that complicate the translated specification. For example, ASN.1 features non-disjoint domains but provides no operations on its values.
- Automated translation into standard LOTOS needs to cope with technical issues like the avoidance of name clashes in the identifiers that it generates. This may lead to a heavy and unnatural style for human reading.
- The translated form ought to be evaluable with the available toolset, which will impose restrictions on the nature of translated types. Since different uses may need different translated forms, the translation should be configurable and extensible.

The translation of a definition in such compact notations may usually be structured in two steps: the first step ensures that a generic type definition for that kind of object is available, either through a library declaration or by explicit definition; the second derives the desired type definition by instantiation and/or renaming of the former. We make this distinction because each step addresses a different problem: the first one is a mean to provide higher order constructs, while the second one gives ad-hoc shortcuts for actualizations and renamings. For example, a tuple notation will be defined as the actualization of a generic tuple of the desired size, which will have to be defined beforehand.

Our approach, presented in the next section, concentrates on the first step. We will discuss in section 5 how it can be combined with other facilities to offer a more complete overall facility.

2.4. A different approach towards data type facilities

Taking into account the arguments given in the preceding section, we have tried to follow a different approach, based on two main requirements.

First requirement: The only public reference language is standard LOTOS.

This implies several things. First, any facility should eventually yield a plain standard LOTOS specification. Since this specification will be the only published part, it should be as close as possible to an ideal human-written specification, from its textual appearance to the organization of its definitions. Second, the specifier himself should have to learn as few non-LOTOS features as possible.

In this spirit, the expressivity gained with compact notations must be weighted against the compatibility lost: verbose standard notations are acceptable as long as they describe clearly application-specific aspects. As a typical example, we will avoid shorthand notations for actualization of common types such as the Set type.

---

[1] [Brinksma92] is in fact oriented towards a revision of the standard language. It proposes more enhancements implying changes in the semantic model itself (most notably, it introduces partial functions).

If we reconsider the two steps discussed at the end of the preceding section in the light of this argument, we essentially need the first one, i.e. the availability of generic definitions for each desired type family, whereas the actualization and renaming step may be left to the specifier.

Second requirement: Software tools are essential to the effective development of LOTOS specifications.

In the first place, the newly defined facility itself should be supported by a tool. Moreover, the specifications produced should allow for the diversity of tools that they will be used with.

The key idea of our proposal is to let the specifier assume that the generic definitions needed are available through a standard LOTOS library declaration. The definitions of different types of a common family share a common identifier pattern with a variant part: for example, types for tuples of different sizes will be identified by "*tuple*<*n*>", where <*n*> is replaced by the number of components in the tuple. Conceptually, such a *virtual library* will contain an infinite number of definitions.

On the other side, an explicit representation of the declared types is needed both for tool environments and for final publication. For this we need a formalism to describe the generic form of identifiers and definitions of a common family (in this case, a collection of macro constructions), and then a software tool to interpret these. Since type definitions are interdependent, the tool must also compute a dependency closure to obtain a closed set of definitions. This will constitute the *real library*, which will be given as input to other tools[1].

It should be noted that the macro extensions will be used in the virtual library specification only. The library user will not see them; he only needs the names and usage of the thereby provided type families. For this purpose a suitably annotated example will generally suffice. The user then requests the types he needs for his particular specification through the standard library declaration mechanism; no language extension is needed at that level.

2.5. A virtual library example

Here is a typical example of how virtual libraries can be used. Suppose the virtual library specification contains the following definition:

```
type  Tuple[n] is
        [i=1..n:Element[i]|,],Boolean
sorts Tuple[n]
opns
        tuple[n] : [i=1..n:Element[i]|,] -> Tuple[n]
[i=1..n:        element[i] : Tuple[n] -> Element[i]
]eqns forall
[i=1..n:        x[i] : Element[i]|,
]
[i=1..n:ofsort Element[i]
        element[i](tuple[n]([j=1..n:x[j]|,])) = x[i] ;
]endtype
```

This definition contains macro extensions, delimited by square brackets and vertical bars. These constructs are defined in section 3.4; let us only mention that the expression "[v=n_1..n_2:*expr*|*sep*]" is expanded into a sequence of *expr* for values of v ranging between n_1 and n_2, separated by *sep*.

---

[1]Though stricto sensu there is only one standardized library, all tools allow the use of user-provided libraries. Since such libraries are not standard their content is supposed to be eventually annexed to the published specification.

In his specification, the user wishes to define a tuple with three components (e.g. source and destination addresses and data). He obtains a generic definition by adding the following declaration in his specification:

```
library tuple3 endlib
```

Then he activates the expansion process to generate a usable library. His specification is scanned, "tuple3" is found and matched against "tuple[n]", resulting in the expansion of the preceding definition for n=3. This results in the following (standard ACT ONE) definition being appended to the generated library:

```
type Tuple3 is
    Element1,Element2,Element3,Boolean
sorts Tuple3
opns
    tuple3 : Element1,Element2,Element3 -> Tuple3
    element1 : Tuple3 -> Element1
    element2 : Tuple3 -> Element2
    element3 : Tuple3 -> Element3
eqns forall
    x1 : Element1,
    x2 : Element2,
    x3 : Element3
ofsort Element1
    element1(tuple3(x1,x2,x3)) = x1 ;
ofsort Element2
    element2(tuple3(x1,x2,x3)) = x2 ;
ofsort Element3
    element3(tuple3(x1,x2,x3)) = x3 ;
endtype
```

This definition may eventually be actualized to obtain the needed type[1]:

```
type Packet is Tuple3 actualizedby Address, Data using
sortnames   Packet      for Tuple3
            Address     for Element1
            Address     for Element2
            Data        for Element3
opnnames    pack        for tuple3
            source      for element1
            dest        for element2
            data        for element3
endtype
```

3. LOTOS VIRTUAL LIBRARIES

In this section we will describe formally the virtual library expansion mechanism. Then we will introduce *VLib*[2], the macro language that has been added to ACT ONE to allow the definition of virtual libraries. Though our discussion will be focused on ACT ONE, we will capture the key components under mathematical abstractions, to put forth the generic aspects of the approach and allow for its application to other languages.

---

[1] Let us recall that, according to the semantics of LOTOS [ISO89], clauses for non-formal sorts and operations in an actualization are handled as renamings.

[2] Pronounce "vee-lib".

We will give syntax definitions using the classical BNF notation, with the following usual extensions: $p \mid q$ for alternatives, [p] for optional parts, { p } for repetitions. Terminal symbols will be surrounded by double quotes. We will also need two auxiliary functions val : { *digit* } $\rightarrow$ *Nat* and *text* : *Nat* $\rightarrow$ { *digit* }, which convert a sequence of digits to its numerical value and vice-versa.

3.1. Virtual libraries and library extraction

Syntactically, LOTOS libraries are sequences of *data-type-definition*, with the following syntax:

data-type-definition ::= "**type**" *type-identifier* "**is**" *p-expression* "**endtype**"

Generally, a type definition is based on other definitions. This dependency relation is called *direct reference* in [ISO89] and corresponds to the following cases:

if *t* is defined as…	then *t* refers directly to…
type *t* **is** *t1*, … , *tn p-spec* **endtype**	*t1*, …, *tn*
type *t* **is** *t0* **actualizedby** *t1*, … , *tn* **using** *repl* **endtype**	*t0*, *t1*, …, *tn*
type *t* **is** *t0* **renamedby** *repl* **endtype**	*t0*

A definition is only meaningful when associated to those it references. We will say that a library is *closed* if all its definitions satisfy this condition. In a virtual library, those definitions are given implicitly using macro constructs. The macro expansion rules, detailed in the next sections, provide a mean to extract the definition for a given name. Then, given a virtual library L and a set of type names N, we obtain a closed real library L' containing definitions for N by extracting from L the reflexive transitive closure of N under the direct reference relation.

Formally, a library is a set of named definitions $\langle name,expr \rangle$, where *name* and *expr* belong to syntactic classes *type-identifier* and *p-expression,* respectively. If L is a library, we have a direct reference relation $\rightarrow_L$ over L and an extraction operation:

$$L{\downarrow}N = \{ \langle name,expr \rangle \mid \langle name,expr \rangle \in L \ \wedge name \in N \}$$

Then the real library extracted from L for a name set N is given by:

$$L{\Downarrow}N = L{\downarrow}anc_L^*(N)$$

where $anc_L^*(N) = \{ name' \mid \exists\ name \in N : name\ (\rightarrow_L)^* name' \}$, with $name \rightarrow_L name'$ iff $\langle name,expr \rangle \rightarrow_L \langle name',expr' \rangle$ for some *expr*, *expr'*.

For the obtained real library to be closed and finite, the virtual library L must fulfil the three following requirements:

(i) L must be *unambiguous*, i.e. all definitions must have different names,
(ii) L must be *closed*,
(iii) L must be *well-founded* w.r.t. direct reference, i.e. there must be no infinite direct reference chain.

(i) and (ii) ensure that $\rightarrow_L$ is correctly defined and (iii) ensures that $L{\Downarrow}N$ is finite. Note that [ISO89] also imposes the existence of a valid dependence order between types, which amounts to say that there is no circular direct reference chain. However the extraction process is not dependant on that requirement.

3.2. Structure of virtual library specifications

We now introduce the language extensions that allow the definition of infinite virtual libraries. Those libraries are organized into a (finite) collection of *type families*, each family

representing a (possibly infinite) collection of types sharing common properties and thus similar definitions. The type names of all members of a family match a common *pattern*, and all the corresponding definitions are instances of a given *template*. Patterns are matched against type names, so that the result of the match, if successful, gives the needed information for the instantiation of the corresponding template. Each family is thus specified as a pair ⟨*pattern, template*⟩.

To allow more flexibility, we require that the syntax for virtual libraries be a superset of that of real libraries, which provides two advantages:

- An existing real library may be included in a virtual library so that its types are available through the extraction mechanism.
- Even with real libraries alone, the extraction mechanism may be used to screen out unused definitions. This may reduce computation costs with some software tools, and this automates the process of building a reference annex containing library types used in a given specification.

This requirement means that VLib specific syntactic elements will be mixed with standard ACT ONE, and confusion between VLib tokens and ACT ONE plain text must be avoided. In this case we have relied on the fact that LOTOS syntax forbids some characters in *p-expression*. We then use those characters ("[", "]", "|") as delimiters for our VLib syntax structures. Several other lexical issues such as letter case conversion or comments handling also arise, but will not be further discussed here.

The LOTOS type families that we define will be indexed by one or more natural numbers. In other words, environments will consist of sets of named natural number values, and will be denoted as assignments e.g. { *x*=1, *y*=5, *n*=10 }. They will be propagated and completed by the template instantiation operator, and will eventually be used in the evaluation of numeric expressions.

This may seem a rather primitive and limited solution. It has the advantage of simplifying the definition and implementation of both the matching and instantiation operations. Moreover it appears that it is sufficient for all classical families that we intend to support with VLib. This argument is further detailed in section 5.

3.3. Patterns for LOTOS type identifiers

VLib patterns are identifiers with place holders, which are themselves identifiers between square brackets. This pattern matches any identifier containing digit strings instead of the place holders, and a match results in the values of those digit strings being associated to the identifiers in the corresponding place holders. For example, "*matrix[i]by[j]*" matches "*matrix3by4*", yielding the environment { *i*=3, *j*=4 }.

More formally, patterns are defined on the following syntax:

pattern	::= *identifier* [*parameters*]
parameters	::= "[" *identifier* "]" [*pattern*]
identifier	::= *letter* { *normal-char* }
normal-char	::= *letter* | *digit* | "_"
letter	::= "A" | … | "Z" | "a" | … | "z"
digit	::= "0" | "1" | … | "9"

The matching operation is a function **match**(*p*,*s*) yielding the environment resulting from the matching of pattern *p* with identifier *s*. It is defined as follows:

if *id* is an *identifier*, *p* is a *pattern* or a *parameters*,
s is a sequence of *normal-char*, *d* is a sequence of *digit*,
E is an environment,

then **match**("","") = ∅

if **match**(*p*,*s*) = *E* then **match**("[" *id* "]" *p*,*d s*) = *E* ∪ { *id*=*val*(*d*) }

if **match**(*p*,*s*) = *E* then **match**(*id p*,*id s*) = *E*

match(*p*,*s*) is undefined in any other case

The last clause indicates that **match** is a partial function: **match**(*p*,*s*) is undefined when *p* does not match *s*.

3.4. Templates for LOTOS type definitions

We will now describe the macro extensions that are used to build the template part of type family definitions. These expressions, when evaluated according to an environment generated by **match**, must return an instance of *p-expression*. Two different approaches were possible for this purpose:

- *Syntax extensions*: The full syntax for LOTOS p-expressions is interpreted, but extensions are introduced to describe the variants across a family: construction of identifiers, variable number of items in sort lists, equation lists, parameter lists, expressions, etc.
- *Macro extensions*: The LOTOS syntax is not covered at all. Instead, instantiation of templates is performed at the character level, without any control that the resulting declaration is syntactically correct.

The first approach may be safer in that it can be defined to always generate syntactically correct p-expressions. However it implies quite a heavy handling, since syntax extensions are needed for every LOTOS syntax class where variations are to be allowed. It is also not very flexible, since variant parts among instances of a template may only be introduced where such extensions have been allowed for.

The second approach is much lighter: a few well chosen elementary constructs can be used to build a wide variety of type definition templates. Since the instantiation occurs at the character level, there is no limit over the LOTOS syntax classes to which these constructs may apply. The computing effort needed is much reduced as well, since hardly any syntactical analysis is performed. Moreover this approach gives control not only on the syntactical but also on the lexical appearance of the final specification, including comments and formatting aspects like indentation and line separation. This is important since the extracted specifications are deemed to be not only input for software tools but also human-readable documents. For these reasons we have followed the latter approach.

We have limited ourselves to some basic constructs without recursion and with only bounded iteration. This ensures that the evaluation of a template will always terminate, and proves sufficient in most cases. The limitations induced by this decision are discussed in section 5.

In informal terms, a template is a piece of plain text scattered with special forms. Such forms are of three types, presented here in a conventional syntax:

- *expressions (expr)* are evaluated according to the current environment and replaced by the text of their value,
- *choices (**if** expr **then** t1 **else** t2)* are replaced by either *t1* or *t2* depending on the value of *expr*,
- *sequences (**for** id=e1 **to** e2 **do** t1 t2 t3 t4 t5)* are replaced by an iterated combination of *t1* to *t5*, evaluated for values of *id* between *e1* and *e2,* as follows:

t1:e1 t2:e1 … t2:e2-1 t1:e2 t3 t5:e2 t4:e2-1 … t4:e1 t5:e1

where *t:v* is the expansion of *t* with *id=v*.

The last form is a versatile refinement of a classical for-next loop. Intuitively, its expansion is a forward sequence of *t1* separated by *t2*, then *t3*, then a backward sequence of *t5* separated by *t4*. The backward sequence proves handy for constructing structured expressions with balanced prefix and suffix, e.g. parenthesized expressions. For simpler iterations only the first one or two arguments are used. A corresponding sequential generation algorithm and some typical examples follow.

```
for id=e1 to e2 do
        expand(t1)
        if id < e2 then expand(t2) fi
od
expand(t3)
for id=e2 to e1 do
        if id < e2 then expand(t4) fi
        expand(t5)
od
```

t1	t2	t3	t4	t5	**for** i = 1 **to** 3 **do** t1 t2 t3 t4 t5
c[i]*	(	x	)	+k[i]	c1*(c2*(c3*x+k3)+k2)+k1
c[3-i]	* (x +		)		c2 * (x+ c1 * (x+ c0))
insert([i],		{}		)	insert(1,insert(2,(insert(3,{}))))
elem[i]	,				elem1,elem2,elem3

The complete syntax of VLib templates is formally defined by the following grammar:

```
template          ::= { text | special-form }
text              ::= { text-char }
text-char         ::= normal-char | "+" | "-" | …
                                        (all legal chars in a p-expression)
special-form      ::= expression-form | choice-form | sequence-form
expression-form   ::= "[" expression "]"
choice-form       ::= "[" expression "?" choice-template "]"
sequence-form     ::= "[" identifier "=" range ":" sequence-template "]"
range             ::= expression [ ".." expression ]
choice-template   ::= template [ "|" template ]
sequence-template ::= template [ "|" template [ "|" template
                        [ "|" template [ "|" template ] ] ] ]
```

This grammar provides no rule for the syntax class *expression*. At this level we may consider the handling of expressions as that of classical numeric and relational operations over natural constants and identifiers. We will assume that a evaluation operator **eval**(*e,E*) is defined that evaluates the numeric expression *e* in the environment *E*.

The instantiation operation is a function **expand**(*s*,*E*) yielding the instantiation of template *t* in environment *E*. This function is defined over an expanded form, which makes explicit some missing optional parts in special forms:

1. missing optional parts of *choice-template* or *sequence-template* occurrences are added, with *template* = *""*,
2. occurrences of *range* of the form *expr* are replaced by *expr ".." expr.*

We may now define the instantiation operation itself as follows:

if *t, t1, …, tn* are *template*,
e, e1, …, en are *expression*,
s1, …, sn are either *text* or *special-form,*
txt is a *text*, *id* is a *identifier*,

then **expand**(*s1 … sn*,*E*) = **expand**(*s1*,*E*) … **expand**(*sn*,*E*)
expand(*txt*,*E*) = *txt*
expand("[" *e* "]",*E*) = *text*(**eval**(*e*,*E*))
expand("[" *e* "?" *t1* "|" *t2* "]",*E*) =
if **eval**(*e*,*E*) = 1 then **expand**(*t1*,*E*) else **expand**(*t2*,*E*)
expand("[" *id* "=" *e1* ".." *e2* "|" *t1* "|" *t2* "|" *t3* "|" *t4* "|" *t5* "]",*E*) = *seq*(*v1*),
where *seq*(*v*) = if *v* > *v2* then **expand**(*t3*,*E*)
else if *v* = *v2* then *t1*:*v* *seq*(*v*+1) *t5*:*v*

else $t1{:}v\ \ t2{:}v\ \ seq(v{+}1)\ \ t4{:}v\ \ t5{:}v$,

$v1 = \mathbf{eval}(e1,E),\ \ v2 = \mathbf{eval}(e2,E),\ \ t{:}v = \mathbf{expand}(t, E \cup \{\ id{=}v\ \})$

3.5. Family specifications for LOTOS libraries

The overall syntax for a virtual library is given by:

family ::= "**type**" *pattern* "**is**" *template* "**endtype**"
virtual-library ::= { *family* }

A given virtual library specification *LS* defines a corresponding library **LIB**(*LS*). The latter contains standard ACT ONE definitions but is purely conceptual since its size is generally infinite.

if *f, f1, …, fn* are *family*,
p is a *pattern,*
t is a *template,*
then **LIB**(*f1 … fn*) = **LIB**(*f1*) … **LIB**(*fn*)
LIB("**type**" *p* "**is**" *t* "**endtype**") =
{ "**type**" *id* "**is**" *p-expr* "**endtype**" | **match**(*p*,*id*) = *E* ∧ **expand**(*t*,*E*) = *p-expr* }

According to its definition, **expand**(*t*,*E*) will in general belong to class *text*: No rule imposes that an instantiation respects LOTOS syntax for *p-expression*. Moreover, **expand** depends on **eval**, which is a partial function (it cannot evaluate variables that are not in the environment). We must therefore impose an external requirement over family definitions, which adds up to the three conditions of section 3.1:

A family "**type**" *p* "**is**" *t* "**endtype**" is *well defined* iff
for any *id* such that **match**(*p*,*id*) = *E* we have
expand(*t*,*E*) is defined and is a *p-expression*

Compatibility with standard LOTOS is ensured by the facts that the syntax classes *type-identifier, p-expression* and *data-type-definition* are included respectively in *pattern, template* and *family,* and that **LIB**(*f*) = { *f* } if *f* is a *data-type-definition*, i.e. the interpretation of a normal type definition is that definition itself.

Finally, the real library extracted from *LS* for a set of type names *N* is given by the function **extract**(*LS*,*N*) = **LIB**(*LS*)⇓*N*. Note that the effective computation of this function requires the interoperation of macro expansion steps and (rudimentary) syntactic analysis steps, to obtain the type dependence information.

4. A PROTOTYPE TOOL

A prototype software tool has been developed, which implements the closed extraction operation on LOTOS virtual libraries, as described in the preceding section. Given a LOTOS specification and one or several virtual libraries, the tool builds the minimal closed real library containing the types mentioned in library declarations of the specification. This implementation is primarily aimed at the validation of the model: the preference has been set on facility of development rather than on performance of the tool, so that in its present form it exhibits a rather poor performance in terms of extraction time.

The prototype has been developed on a UNIX workstation, using general-purpose text-processing tools like *awk*, *lex* and *m4*. The processing is distributed among several modules: extraction of type dependency information, name matching and pattern instantiation. These modules are implemented as separate UNIX processes and interact through text files or UNIX pipes.

The algorithm used for the closed extraction is given hereafter in Pascal-like syntax. This algorithm is optimal in that it performs the minimal needed number of direct extractions, and it always terminates, provided the library is well-founded.

```
function extract( L : Lib, N : ℘ Name ) : Lib;
var
        L0 LN : Lib ;
        N0, N1, NN : ℘ Name ;
begin
        N0 := N ;
        NN := ∅ ;
        LN := ∅ ;
        while N0 ≠ ∅ do begin
                L0 := L↓N0 ;
                LN := LN ∪ L0 ;
                NN := NN ∪ N0 ;
                N1 := ref(L0) ;
                N0 := N1 - NN ;
        end ;
                extract := LN ;
end ;
```

Two functional extensions are also offered with respect to the extraction principle presented above. The first one allows every type definition of the LOTOS specification to be searched, while the second one allows a full real LOTOS library to be included in the built library. In both cases, both defined and referenced type names are captured so that, on one side, all undefined referenced types will be extracted and, on the other side, already defined types will not be extracted again.

The separation into several processes allowed each module to be developed with the most appropriate UNIX tool. The main source of overhead in the prototype is due to the fact that, for each iteration of the algorithm, a new pattern matching process is started and all needed data are read again. More efficient implementations are possible but would have required a much deeper programming effort.

A basic virtual library has also been written. Besides all types of the standard library and a few other isolated data types such as integers or finite maps, it provides family definitions for the following data structures:

- *Element[n]*: a renaming of the standard type Element, to be used by other parameterized types that need several formal sorts.
- *Tuple[n]*: the *n*-field record structure already described, with operations for assembling a tuple, accessing and modifying components.
- *Enum[n]*: a finite enumerated type with *n* values. Cyclic successor and predecessor operations are provided, as well as mapping to natural numbers.
- *Union[n]*: an approximation of the disjoint union of *n* sorts, with explicit conversion operations to map member sorts to the union sort and vice-versa (the latter are partially defined).
- *Modulo[n]*: numbers with modulo-*n* arithmetic. Provides addition, subtraction and product, and mapping to and from natural numbers.
- *Table[n]*: an *n*-column table, defined as a set of *Tuple[n]*. In addition to those offered by those two types, operators are provided for filtering, modifying and searching according to a given column.

5. POSSIBLE EXTENSIONS

We have argued above that most syntactical extensions are not worth the loss of LOTOS compatibility or translation effort that they entail, if a sufficiently complete set of generic

types can be provided through a VLib-like mechanism. It is however interesting to study how both solutions could be combined.

Operation syntax extensions like mix fix notations (as in array indexing *x[i]*) or variable argument lists (as in set denotations *{a,b,c,d,e}*) are usually not used because they require too much translation effort. Indeed, the translation must occur not only in type definitions but also in all data terms throughout the definition. Furthermore, they should be supported "at run time" in every tool that reads or displays data terms. Since they do not affect type dependencies, they could be used in the specification without affecting VLib (as long as no syntax overlap arises). If they are to be used in libraries too then they should be processed in the output of VLib .

On the other hand, type definition ad-hoc notations for classic types (e.g. for tuple definitions) are the common ones because they correspond to the strongest need and require no "run-time support" as in the previous case. They will however need a more careful treatment w.r.t. VLib because they may hide some type dependency information: their expansion may (and probably should) reference library types. Therefore, those notations would have to be expanded in the specification before it is scanned by VLib. If such notations are desired in the library as well, then they have to be expanded after each direct extraction of the library, to reveal ancestor types.

Mainly for simplicity of implementation, we have limited the available structures both at the pattern and template level to a few simple general ones. Though they appear to be sufficient for specifying most needed type families, we will now envisage several more flexible extensions.

The pattern/template family scheme could also be applied to other elements such as sorts or operations. A typical example of this would be the definition of numeric constants[1] (1, 2, 10, 256, …) within an extended natural number definition: the extracted type should then contain nullary operations (and corresponding equations) for each such value used in the specification. The provision of such a facility however proves difficult because operations do not match the virtual library model and this would make the closure computation considerably more difficult, if not impossible. Moreover we may use a work-around by transforming the operation family into a type family, so that each family member defines one operation.

It might also be useful to have a way of defining recursive template parts. For example, for tool performance reasons, we might prefer to define numeric constants in polynomial terms, e.g. 345 = (3*10+4)*10+5. The generation of such an equation for any value *n* could be defined as a recursive function *poly*(*n*):

*poly(x) = [x < 10?[x]|([poly(x div 10)])*10+[x mod 10]]*

Such a facility would augment the expressive power, but would also increase the complexity of the instantiation module and would allow the definition of infinite templates (i.e. non-terminating instantiations). Again a work-around is possible in some cases by using a chain of dependent types instead of the recursive function: for the preceding example, a type *Nat345* would be based on types *Nat34*, *Nat10* and *Nat5*.

More flexible possibilities could also be offered for the variable parts in patterns: more general strings could be matched. This would require more complex pattern matching, environment handling and expression evaluation algorithms. Furthermore it seems that most possible uses of such a functionality could be easily and more clearly handled using the current definition. Typically, such non-numeric arguments would be used to pass (parts of) sorts or operations identifiers, which may instead be instanciated through standard actualization constructs. An admittedly worthwhile extension would be the support of other numeric bases such as hexadecimal numbers.

---

[1] Let us remind that constants are defined in ACT ONE as nullary operations (e.g. *opns 0 : -> Nat*).

6. TOWARDS A SAFE ADT SPECIFICATION DISCIPLINE

There remain other difficulties such as ensuring conservative type definitions or dealing with specificities of other tools. In this last section we would like to propose an ACT ONE specification discipline in order to minimize such risks:

- Virtual libraries are carefully designed by ACT ONE experts to ensure that they exhibit as much as possible all the desired properties. They must contain a sufficiently various set of basic types to allow the construction of most particular data structures that will be encountered in any situation.
  If other software tools impose restrictions on the specifications, then the experts will provide a version of the library tuned for those tools, and will ensure that it preserves as much as possible the semantics of the original version, and indicate clearly where and why it does not.
- The specifier deliberately restricts himself from specifying new types from scratch, but instead defines its types exclusively by (repeated) actualization and renaming of library types. Definition of pure enrichments[1] might also be allowed.

Since actualization and renaming will not introduce new difficulties if the involved types are sane, this discipline reduces the risk of obtaining ill-behaved definitions. Defining new operations is more risky, but it is still easier to assert properties in this restricted case than for the definition of a new type from scratch. We believe this approach is largely unrealistic with normal libraries, but may become tractable with an extension like VLib providing a more comprehensive set fundamental types such as tuples or enumerations.

7. EVALUATION AND CONCLUSIONS

At the theoretical level, the virtual library approach requires a minimal extension to the standard LOTOS definition. The only change is that the LOTOS library is extended to contain an infinite number of definitions, otherwise leaving the language intact. All the specifier needs to know is the kind of definitions and associated identifiers available in the new virtual library. For this purpose, a natural language presentation of each provided family with a typical example will usually be clear and precise enough: he need not learn any extended syntax for using the new facilities or understanding their meaning. Indeed, the VLib syntax and translation are only required for tool support: if no tool is taken into account, no concrete language is really necessary, though a notational convention is still needed to describe the infinite type families in finite terms.

On a more practical point of view, using virtual libraries in software-supported specification developments will require the use of an extraction tool such as the prototype described in the preceding section. Such an extraction will have to be repeated during the construction of the specification, to ensure that the extracted library used by the other tools follows the evolution of that specification. Note however that a new extraction will be required only upon changes in the library declarations of the specification, whereas other data type extensions usually require that the translation be done on each modification of any part of the specification, which will occur much more frequently.

Since the definitions are instanciated from templates defined in a formal language, it is easy to adapt these templates to provide distinct versions of the virtual library for different environments, and in particular for subsequent processing by different tools. This distinguishes our approach from other extension tools that have their translation built into program code, which makes such adaptations much more difficult.

---

[1] An enrichment is a conservative extension without new sorts, i.e. a specification of operations with defining equations so that they do not introduce any new data values nor equate any previously defined values [Ehrig85].

Globally, VLib has succeeded in filling its niche: it provides a lightweight but flexible mean of offering the kinds of generic type definitions that everyone wants but the classical libraries do not provide. Assuming that LOTOS remains the only reference, it promotes compatibility against conciseness and introduces no new notation at the specification level. Though this is far from solving all LOTOS data type weaknesses, we believe it is a quite reasonable solution until a new improved version of the language is standardized… and tools are available that support it.

ACKNOWLEDGEMENTS

The original ideas presented here are largely the fruit of my interactions with Eric Lallemand during the design and implementation of his tool DAFY. I also want to thank the anonymous referees for their contribution to the quality of this final version of the paper.

REFERENCES

[Alderden89] Alderden R.B.
"Functionality of an ADT interpreter"
in: Eijk P.H.J.v., Vissers C.A., Diaz M., The Formal Description Technique LOTOS : results of the ESPRIT/SEDOS project, North Holland, Amsterdam, 1989, ISBN 0-444-87132-2.

[Bochmann89] Bochmann G. v., Deslauriers M.
"Combining ASN.1 support with the Lotos language"
in: Brinksma E., Scollo S., Vissers C.A., eds., Protocol Specification, Testing and Verification IX, North Holland, Amsterdam, 1989.

[Boehm89] Boehm P., deMeer J., Schoo P.
"PERLON Persistency Checker for Data Type Definitions"
in: Eijk P.H.J.v., Vissers C.A., Diaz M., The Formal Description Technique LOTOS : results of the ESPRIT/SEDOS project, North Holland, Amsterdam, 1989, ISBN 0-444-87132-2, pp. 285-302.

[Brady90] Brady F., Boshier A.G., Szczygiel B.M.
"One2One - A tool for translating ASN.1 to ACT ONE"
in: Quemada J., Mañas J., Vazquez E., eds., Formal Description Techniques III, North-Holland, Amsterdam, 1991, ISBN 0-444-89024-6, pp. 539-542.

[Brinksma92] Brinksma E.
"Enhancements of LOTOS"
in: Proceedings of the Third Lotosphere Workshop and Seminar, Pisa, Sept. 1992.

[Eertink92] Eertink H., Wolz D.,
"Symbolic Execution of LOTOS Specifications"
to be published in: Diaz M., Groz R., eds., Formal Description Techniques V, North-Holland, Amsterdam, 1993.

[Ehrig85] Ehrig H., Mahr B.
"Fundamentals of Algebraic Specification 1"
EATCS Monographs on Computer Science, Vol. 6, Springer Verlag, 1985, 321 p.

[Eijk91] Eijk P.H.J.v.
"Tools for LOTOS, a Lotosphere overview"
in: tutorial proceedings of 11th Symposium on Protocol Specification, Testing and Verification, Sydney, 1991.

[Garavel89] Garavel H.
"Compilation of LOTOS Abstract Data Types"
in: Brinksma E., Scollo S., Vissers C.A., eds., Protocol Specification, Testing and Verification IX, North Holland, Amsterdam, 1989.

[ISO89] ISO/IEC JTC 1/SC 21/WG 1/FDT/C
"Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour"
ISO IS 8807, 1989.

[Lallemand91] Lallemand E., Leduc G.
"A LOTOS Data Facility Compiler (DAFY)"
to be published in: Rose G., Parker K., eds., Formal Description Techniques IV, North-Holland, Amsterdam, 1992.

[Manas88] Manas J., de Miguel T.
"From LOTOS to C"
in: K.J. Turner, ed., Formal Description Techniques, North-Holland, Amsterdam, 1989, ISBN 0-444-87126-8, pp. 79-84.

[Scollo86] Scollo G.
"Some Facilities for Concise Data Types in LOTOS"
Esprit P410 SEDOS/C1/WP/13/T, 17 March 86, 52 p.

[Thomas92] Thomas M.
"A Translator for ASN.1 into LOTOS"
to be published in: Diaz M., Groz R., eds., Formal Description Techniques V, North-Holland, Amsterdam, 1993.