Non-standard Interpretations of LOTOS Specifications[1]

Tommaso Bolognesi *
Diego Latella *
Elisabetta Zuppa **

* CNUCE / C.N.R. - 36, Via S. Maria - 56100 Pisa - ITALY
phone: +39-50-577.201 - fax: +39-50-576.751
e-mail : bolog *or* latella@ fdt.cnuce.cnr.it

** Consorzio Pisa Ricerche
9, Via Risorgimento - 56100 Pisa - ITALY
phone: +39-50-500.995

Abstract

Non-standard interpretations of LOTOS specifications are proposed as a most convenient and conservative way to extend the expressivity of the language without affecting its standard syntax and transition-system-based semantics. Some simple non-standard interpretations, alse called *view functions*, are introduced. Two different styles of formal definition are adopted (denotational and operational) for providing, respectively, a refinement of the standard LOTOS process functionality parameter, and a new parameter measuring the degree of synchronization exhibited by a specification.

INTRODUCTION

A LOTOS [ISO89a, BB87] specification is meant to capture, a priori or a posteriori, some properties of, typically, a concurrent and reactive hardware/software system (Figure 1.1).

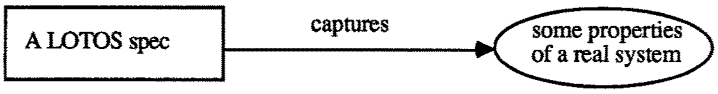

Figure 1.1 - Describing a real system in LOTOS

The main motivation of our work is found in the observation that the expressive power of standard LOTOS is inadequate for capturing a variety of desirable system properties. LOTOS is meant to describe temporal orderings of events. The standard LOTOS semantics is based on Labelled Transition Systems

1) This work has been carried out within the ESPRIT-LOTOSPHERE Project

(LTS); a LOTOS specification ultimately describes a system's behaviour as a tree of events that unrolls along the time axis (without involving explicit time values). Consider now, for example, properties like functional behaviour, physical distribution of functionality, time demands and ergonomics of user-interface.

In virtue of the LTS-based LOTOS semantics, we can safely adopt LOTOS for capturing the functional behaviour of a system, based on the assumption that it can be expressed in terms of sequences of events. However, the language is unsuited for characterizing properties 2, 3, and 4: the standard LTS derived from a given LOTOS specification does not provide any information on the physical distribution of functionality, on the timing of events, or on the ergonomics of the user-interface.

Several extensions to LOTOS have been proposed in recent years. Syntactic extensions, offering alternative syntactic forms (such as the G-LOTOS graphical syntax) or short-hands, attempt to provide more convenient notations, without altering the underlying semantic model. Semantic extensions include new behavioural operators, notions of structured events or explicitly timed events, and the module concept, and are meant to offer more expressive underlying semantics.

Clearly we cannot hope to express more properties in the list above by adopting pure syntactic extensions: enhancements are needed at a semantic level. On the other hand LOTOS is today an international standard, whose development took almost a decade; it is quite clear that stability is now a key requirement for the wide-spread acceptance and application of the language. The main purpose of this paper is to propose a solution to these two apparently conflicting needs. Such solution is based on the idea that the enhanced expressiveness should not be achieved at the cost of replacing or modifying the existing standard syntax and semantics, but it should, rather, *co-exist* with them. By viewing the standard semantics of LOTOS as an interpretation function that maps the LOTOS syntax into the domain of LTS's, our approach essentially consists in adding to the standard LOTOS semantic interpretation some non-standard (semantic) interpretations, called *view functions*, also applied to the standard LOTOS syntax. To our knowledge, none of the extensions proposed for LOTOS so far falls under such scheme. We believe, instead, that our "multi-semantic approach" bears the promise of offering to the LOTOS users a wide selection of expressive tools (the view functions) without jeopardizing the stability of the standard language, and deserves the attention and contribution (in terms of new view functions) of the LOTOS community.

The paper is organized as follows. In Section 2 we discuss the notion of compatible extensions to LOTOS, and present a classification of such extensions. In Section 3 we concentrate on extensions based on non-standard interpretations and present some simple examples. Sections 4 and 5 introduce two different techniques (denotational and operational approaches) for the definition of view functions. Such techniques are applied, respectively, for (re-)defining the LOTOS process functionality parameter, and a newly introduced view function called SyncDegree (technicalities are found, respectively, in Appendices A and B). In the conclusive Section 6 we discuss how the ideas presented in the paper can be put to work in the framework of a LOTOS-based software development environment, and present some preliminary implementation experiences. Some familiarity with LOTOS (processes) is assumed.

COMPATIBLE LOTOS EXTENSIONS

We discuss here, at an abstract level, three different ways to conceive extensions to the LOTOS language (see Figure 2.1). They are called *compatible* extensions because in all three cases some syntactic and semantic elements of the standard language are preserved. (Indeed, based on such definition of compatibility, any non-compatible extension to LOTOS should be considered as a new language, that may at most claim to have been *inspired* by LOTOS; thus the attribute "compatible" is a bit redundant).

The starting point of any extension is the LOTOS standard [ISO89a], which consists of the three elements shown in Figure 2.1.a:

- the standard *syntax*,
- the standard semantic *model* (LTS's plus, possibly, an equivalence relation) and
- the standard semantic *interpretation*, which associates a semantic model to syntactic objects (specifications).

Syntactic extensions (Figure 2.1.b)

Shorthands are syntactic extensions that are meant to provide abbreviations for cumbersome constructs or combinations of constructs of the standard syntax. The *expansion* of a shorthand returns the original, cumbersome notation. An example of LOTOS shorthand is:

MaxCoop {P1[G1], ..., Pn[Gn]}

proposed in [B90], which denotes a behaviour expression (indeed, many equivalent ones) where n processes P1, ..., Pn, with associated actual gate sets G1, ...Gn, are composed by multiple occurrences of the (binary) parallel operators "|[...]|" or "|||".

Alternative concrete syntaxes are meant to provide alternative representations of the constructs of the language, and enhance the readability of LOTOS specifications. An example of such an extension is the G-LOTOS (for "graphical") syntax being standardized in ISO [ISO89b], which is meant to provide first glance recognizability of the fundamental structure of a specification, and of its behaviour expressions in particular. G-LOTOS provides simple and well distinguishable graphic representations for sequentiality, parallelism, and nondeterminism. An alternative syntax is typically provided with a translation function that maps it onto the standard syntax, so that the former inherits the semantic interpretation (and model) of the latter.

In conclusion, shorthands and alternative syntaxes do not affect the semantic level of the language.

Upward compatible extensions (Figure 2.1.c)

Such extensions are meant to enhance the expressiveness of the language at the semantic level. First a desirable extension of the underlying semantic model is identified, and then further syntactic constructs and an associated extension of the semantic interpretation function are introduced, that "cover" the extended semantic model. "Upward compatibility" means that the extended interpretation is conservative: its restriction to the standard syntax coincides with the standard interpretation, thus it yields instances of the standard model.

As an example, consider a LOTOS extension where events are no longer atomic, but composite objects such as sets. Synchronization may then take place based on the non-empty intersection of two events, and hiding can be applied to just a subset of an event (partial hiding). Still, it may be possible to design such new features in such a way that the standard syntax and semantics are special cases of the extended ones: a standard event is a singleton, standard hiding is partial hiding applied to the complete event, and so on. A proposal for LOTOS extensions of this kind is found in [B88].

Another interesting branch in the research on LOTOS extensions deals with the introduction of time parameters (see, among others, [B88, HTZ89, QAF89, BLT90]). The question on whether such extensions are really upward compatible (is the new semantic model a proper superset of the standard one ?) is not an easy one, and we do not discuss it here.

Finally, the introduction of *specification modules* [B89] as a means for the decomposition of a specification into a set of distinct, self-standing documents has to be mentioned here. In fact, it allows for a better management of specification development activities, still keeping compatibility with the LOTOS existing features for hierarchical specification.

While in principle upward compatibility is very elegant and appealing, since, in some sense, it preserves the standard syntax and semantics of the language, in practice it still seems to conflict with the need of stability of the standard language: the user of the extended language, which does want to take advantage of the enhanced expressivity offered by it, is essentially confronted with a new language.

The *multi-semantic* approach (Figure 2.1.d).

Rather than enhancing the standard language by conservatively extending its syntax and semantics, the *multi-semantic* approach leaves the syntax unchanged, and offers further, non-standard interpretations of it that are simply added to the standard one. The new, non-standard interpretations, also called view functions, are meant to exploit the same syntax for capturing various aspects of the specified system that are out of the reach of the standard, LTS-based interpretation.

While on one hand such an approach offers enhanced expressivity at the semantic level, on the other hand it is *truly* conservative, and by no means destabilizes the standard; the approach is further discussed in the next section.

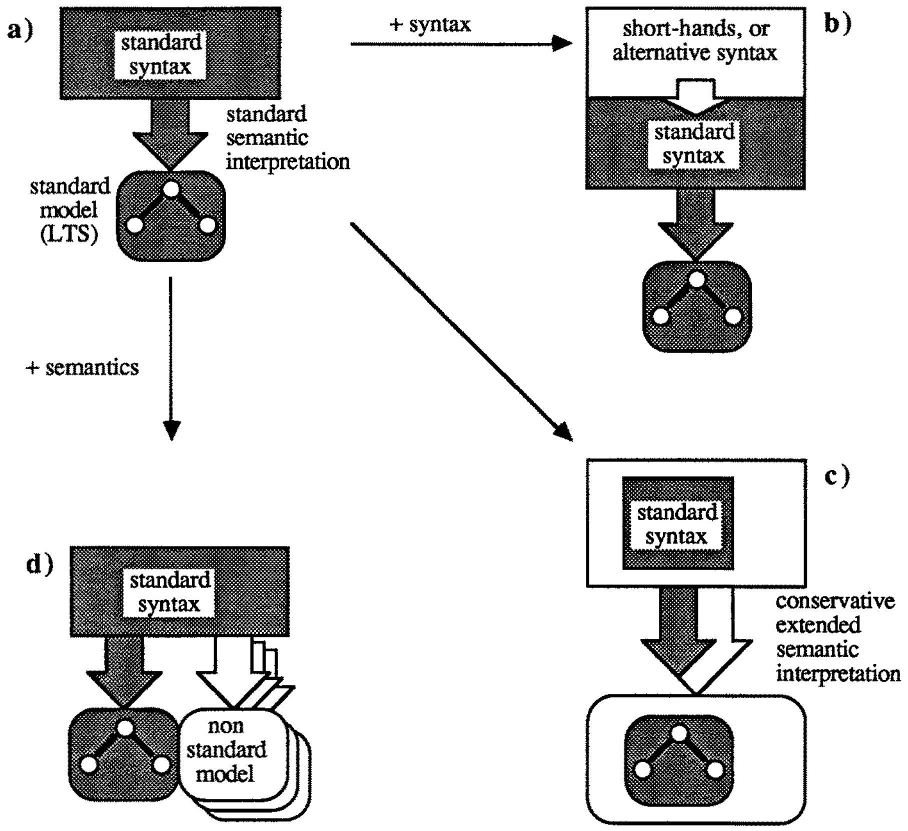

Figure 2.1 - Compatible LOTOS extensions

VIEW FUNCTIONS

Figure 3.1 is a refinement of Figure 1.1 which illustrates more precisely the sense in which a LOTOS specification describes a real system. Property capturing is only possible by means of interpretations of the specification, that map it into mathematical objects, or models. Figure 3.1 is also a refinement of Figure 2.1.d which illustrates the multi-semantic approach.

Functional properties are modeled via the standard, LTS-based semantic interpretation.

Non-functional properties require the ad-hoc definition of non-standard interpretations, also called *view functions*.

Further properties exist that remain unspecified formally.

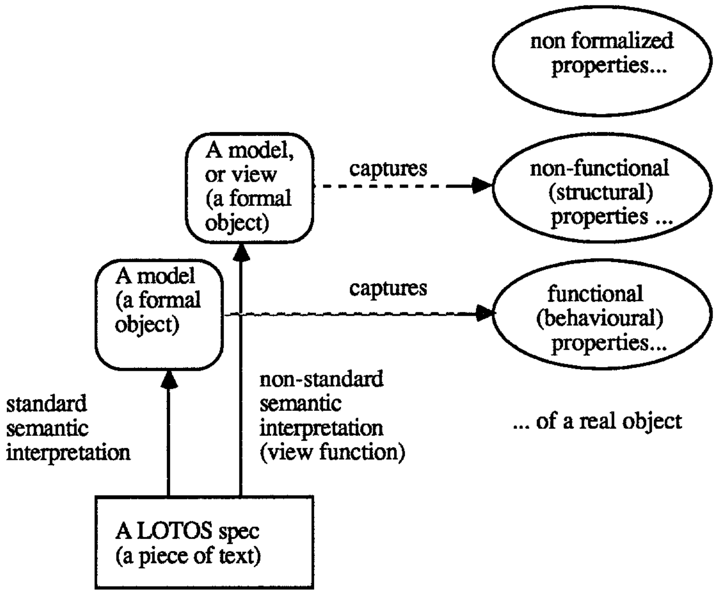
non formalized properties...
A model, or view (a formal object)
captures
non-functional (structural) properties ...
A model (a formal object)
captures
functional (behavioural) properties...
standard semantic interpretation
non-standard semantic interpretation (view function)
... of a real object
A LOTOS spec (a piece of text)

Figure 3.1 - Property capturing by standard and non-standard interpretations

For example, with respect to the four properties listed in Section 1, we may

- rely on the standard LTS for modelling the functional behaviour of a system (e.g., for stating formally that a ConnectRequest event is followed either by a ConnectConfirm or by a DisconnectIndication);
- adopt a specific view function for modelling the physical distribution of functionality (e.g. for stating formally that the system is decomposed into a Caller, a Responder, and a Medium, that must be interconnected according to the well known pattern); and
- leave aspects such as the timing of events and ergonomics of user-interface not formally specified.

Thus, a view function maps LOTOS specifications (or parts of it) into a suitable set of mathematical objects, called *view domain*. The application of a view function to a specification yields an element of the relevant view domain, called *view*.

In principle one may define very complex view functions, perhaps for capturing also *functional* properties that escape the modelling capabilities of the standard interpretation. Thus, the correspondence between standard (resp. non-standard) interpretations and functional (resp. non-functional) properties, as implied by Figure 3.1, should not be taken as too rigid. Indeed, even the distinction between functional and non-functional properties is not commonly agreed upon as one might wish.

Without further discussing terminological issues, we simply observe that, in practice, views are *not* complex, possibly infinite objects with functional-semantic flavour, such as *failure sets* [BHR84]; they are rather simple, finite objects, such as a natural number, a finite set of LOTOS gates, or of LOTOS behavioural operators. Moreover, most of them are understood as conveying information on the *structure* , rather than the functionality or behaviour of the specified system.

The argument of a view function depends on the particular view one is interested in, and may be either a *process definition* or a *behaviour expression*. Of course, views on *process definitions* will make reference to the defining *behaviour expressions* of such definitions. We shall let Bex denote the set of LOTOS behaviour expressions. For the illustrative purposes of this paper, we restrict to Basic LOTOS, where data type definitions and value expressions are not present. Furthermore, we shall make reference to *Flat* process definitions of the type:

process P [g_1, ..., g_n] F := B_P
where **process** P1 ... :=B_{P1} **endproc**
...
process Pk ...:= B_{Pk} **endproc**
endproc

F stands for *P's declared functionality* (either **exit** or **noexit**). The *behaviour* of *P* is defined by its top-level *behaviour expression Bp* (also called "defining behaviour expression") which, in turn can make reference to processes *P1,...,Pk*. The definitions of the Pi's processes do not include, in turn, process definitions, that is, they have no "where" clause and, in general, mutually recursive. Also, when no ambiguity may arise, we shall avoid to make explicit reference to the environment defined by the definitions of processes *P1,...,Pk*.

Some view functions are informally introduced below, by means of examples. In describing them, we shall refer to the variables that appear in the process definition scheme above.

Given expression BP, *ObsGates (BP)* identifies the set of gates where process P may be active. For example, *ObsGates(a; stop [] b; P[c, d]) = {a, b, c, d}*. An interesting extension of the ObsGates is the *TypedGates* view function, formally defined for full LOTOS in [BBFN90]. The typed gates of BP are, again, the gates where process P may be active, but also include information on the sorts of the (tuples of) data values that may be exchanged at those gates.

A *gate structure* is a set of gate sets; let *GateStructures* be the set of such objects (GateStructures = $2^{2^{\text{Gates}}}$). The *GateStructure* view function is a kind of refinement of the ObsGates function, that is sensitive also to the parallel substructure of the given behaviour expression BP: it is meant to provide a family of gate sets, each corresponding to one of the parallel components of BP. Some examples follow:

GateStructure(a; stop [] b; P[c, d]) = {{a, b, c, d}}.
GateStructure(a; stop ||| b; P[c, d]) = {{a}, {b, c, d}}.

The GateStructure view function represents a good example of what we mean by extracting from a specification (or, rather, a process definition of P) some information that is not provided by the standard semantic interpretation. Clearly a labelled transition system is a "flat" object that does not show the partitioning of the gate space corresponding to the parallel sub-processes of P.

We are not interested here in a detailed presentation of several view functions, and on motivating each one of them individually. At the general level, we have already advocated the notion of non standard interpretation as a means for extending the expressivity of LOTOS in a most conservative way. The application of the newly introduced concept to the definition of *correctness preserving transformations* is thoroughly discussed in [BBFN90]. In that document, *correctness preserving transformation* problems are decoupled into (semantic) *correctness preservation requirements*, which deal with *standard interpretations*, and *transformation requirements*, dealing with *non standard interpretations*.

In the rest of the paper we concentrate, instead, on two alternative approaches to the formal definition of view functions: *denotational* and *operational*. The former is applied to (a refinement of) the definition of the LOTOS *process functionality*; the latter is used for defining a new view function, called *SyncDegree*.

DENOTATIONAL APPROACH TO THE DEFINITION OF VIEW FUNCTIONS

In this section we shall follow the denotational approach for the definition of view functions. The denotational approach has been introduced by Scott and Strachey [S70] as a means for defining the semantics of programming languages. The approach consists in providing a *direct mapping* of language constructs into the mathematical objects they *denote*, like sets and functions. A suitable *interpretation function* must be defined for each syntactic category of the language. Such a function will map the syntactic category into its *semantic domain*, giving thus *meaning* to the constructs in that category. Some special technicalities are needed for handling recursion: it is required that semantic domains be *complete partial orders* (c.p.o.) rather than unstructured sets, and that functions be *continuous* over them. When such requirements are fulfilled, standard fixpoint theory results can be used. The reader is referred to [S77] for a comprehensive description of the approach.

The denotational approach has been devised for giving language constructs their "standard" interpretation, also called *dynamic semantics* because it accounts for actual program execution. However, a major advantage of the method is that, given the high degree of freedom in the choice of semantic domains, one may similarly define *non-standard* interpretations of the language, including *static semantic* aspects. For example, one may define *Type* interpretation functions, which map into a domain of types, thus providing a mathematical framework for type-checking. Other interpretations may be defined for supporting *strictness analysis*, which gives information about the behaviour of programs with respect to termination in the context of functional programming languages [CPJ85].

A general scheme for the denotational definition of view functions

The two fundamental syntactic categories of *Flat* Basic LOTOS are *ProcDefs* for *process definitions* and *Bex* for *behaviour expressions* (see the process definition scheme in Section 3). Thus, for a given *view domain*, i.e. non-standard semantic domain *V*, we have to define two different interpretation functions. One of them, *BexV*, gives meaning to *behaviour expressions* and has the following type:

$$\mathrm{BexV} : \mathrm{Bex} \rightarrow \mathrm{V}^k \rightarrow \mathrm{V}$$

Given a *behaviour expression* B which, in the general case may contain instantiations of the local *process definitions* occurring in the top level process definition, *BexV[B]* takes the *meaning* of such definitions, represented by a k-tuple $v_1,..,v_k$, as parameter and returns the meaning of B. Thus, in order to determine the interpretation of a *behaviour expression* we need the interpretations of all the process definitions appearing in the top level definition. This is done by means of the *PDefsV* interpretation function, the type of which is the following:

$$\mathrm{PDefsV} : \mathrm{ProcDefs}^k \rightarrow \mathrm{V}^k$$

PDefsV takes the k-tuple of *process definitions* occurring as local definitions in the top level process definition and returns the k-tuple of their interpretations.

$PDefsV[PD_1,..,PD_k]$ will in turn be defined in terms of *BexV* which will give meaning to the defining behaviour expressions $B_1,..,B_k$ of $PD_1,..,PD_k$. Now a problem arises; some of the B_i s may contain recursive instantiations of some PD_js. So, $BexV[B_i]$ must be provided the meaning of $PD_1,..,PD_k$ which is just the object we are trying to define! This induces a recursive structure on the definition of $PDefsV[PD_1,..,PD_k]$ itself which must indeed satisfy the following equation:

$$PDefsV[PD_1,..,PD_k] = <BexV[B_1](PDefsV[PD_1,..,PD_k]),...,BexV[B_k](PDefsV[PD_1,..,PD_k])>$$

Under proper assumptions, namely that V is a c.p.o. and that *BexV[B]* is continuous over V^k the above equation has solutions; such equation is written in more readable form as a system of k equations in k variables ranging over V:

$v_1 = BexV[B_1]v_1v_2..v_k$
$v_2 = BexV[B_2]v_1v_2..v_k$
...
$v_k = BexV[B_k]v_1v_2..v_k$

We *define* $PDefsV[PD_1,..,PD_k]$ as follows:

$$PDefsV[PD_1,..,PD_k] = \mu(v_1..v_k).<BexV[B_1]v_1..v_k ,..., BexV[B_k]v_1..v_k>$$

which is the *minimal* solution of the above system of equations in the partial order V^k.

We are now ready for defining the view we are interested in on the top level process definition PD. It will be given by the application of the *VInt* -erpretation function to PD, where *VInt* will be defined in terms of *BexV* and *PDefsV* :

VInt[**process** P [$g_1,...,g_{1n}$] : F := B **where** $PD_1,..,PD_k$ **endproc**] =
    $BexV[B](PDefsV[PD_1,..,PD_k])$

In the following we shall apply the denotational framework to the definition of the *functionality* parameter. In the case of Basic LOTOS, the *functionality* parameter reduces to the *possibility* of *successful termination* of a *behaviour expression.* The formulation of the problem that the view would be requested to answer is then: "Is there *any* computation, among those derived from a behaviour expression, which successfully terminates?" or, more formally: "Is there a path starting from the root of the labelled transition system associated to the behaviour expression which leads to a δ-transition?".

It is well known that such a property is undecidable ([ISO89a]) so, there is no hope to define a computable interpretation function which decides it in all possible situations, i.e is *total,* and which gives *correct* and *complete* information. So, apart form giving an effective definition of a *partial* function which returns an answer when applied to a possibly successfully terminating *behaviour expression* and which may be undefined otherwise, the best we can do is to effectively define an *appropriate* abstract interpretation function which only *approximates* the functionality property, namely a function which enjoys the following properties:

i) It *must* be 'safe' in the sense that it must never suggest that a *behaviour expression cannot* terminate successfully when the associated labelled transition system *does* contain a δ-transition.

ii) It *should* be as 'informative' as possible, that is it should detect in as many cases as possible the *impossibility* of termination of *behaviour expressions*, namely the fact that the associated labelled transition systems *does not* contain δ-transitions, .

Using the notation introduced in [ISO89a] we shall say that the functionality of a process is <> iff there is a path starting from the root of its labelled transition system which leads to a δ-transition and it is *0* otherwise. We define the *Func* set as {0,<>}. Safety implies that, for any process P:

FuncInt[P] = <>	iff *maybe there is* a path starting from the root of P's labelled transition system which leads to a δ-transition, and
FuncInt[P] = 0	iff *there is definitely* no path starting from the root of P's labelled transition system which leads to a δ-transition

Now, a function *FuncInt* such that *FuncInt[P]* = <> for all P would be safe, but it would be of no help! So, the problem is to find a definition for *FuncInt* which is safe but also sufficiently informative. There is not an obvious solution to such problem. For instance, one possible solution could be a modification of the standard functionality function *func* defined in [ISO89] obtained by simply removing all the information related to LOTOS types. A consequence of such a choice would be that a process definition like the one given below would be assigned a <> functionality, while it is obvious that any instantiation of process P will definitely fail to terminate!

process P [g] : **exit** := g ; P[g] **endproc**

For the same reason, the standard functionality assigned to P[g] >> **exit**, where P is the process defined above, is <> while *0* would be more appropriate. The only way out from such situations is to *compute* the functionality of the body by means of a safe interpretation function and define the functionality of the instantiation to be equal to the result, possibly checking the declared one for consistency .

In Appendix A a safe functionality interpretation function is defined which is a bit more informative than the standard one; for instance, it assigns *0* to the *behaviour expressions* of the above examples.

OPERATIONAL APPROACH TO THE DEFINITION OF VIEW FUNCTIONS

In this section we shall follow the operational approach for the definition of view functions. The operational approach has been used in the standard definition of the LOTOS language for defining its dynamic semantics. The approach consists in providing a derivation system for transitions that is used for the definition of labelled transition systems.

The derivation system consists of axioms and inference rules that can be used for deriving the transitions associated with a given behaviour expression, based on its syntactic structure. For an introduction to structured operational semantics, see [P81].

The standard interpretation of LOTOS does not give explicit relevance to all the possible aspects of a specification or process in which one can may be interested. However, it may happen that a relatively small enrichment of each individual axiom and rule of the semantics be sufficient for defining interesting view functions and capturing some of these aspects. An advantage of such an approach is that it preserves the structure of the existing standard dynamic semantics. On the other hand, the transition system associated with a behaviour expression may be infinite: this implies that, in some cases, the computation of the view function may diverge, and that the function is only partial. This is what happens with the SyncDegree view function defined below.

Consider the labelled transition system LTS(BP) associated to the defining behaviour expression *BP* of a process definition *PD* of process *P*. The arc-labels on the LTS(BP) do not provide information on *how* the actions are performed. For example, the two expressions:

B1 = a ; **stop**   and   B2 = (a ; **stop** |[a]| a; **stop**)

are associated to the same labelled transition system, shown in Figure 5.1.

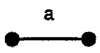

Figure 5.1 - A Labelled Transition System

However, the *a* action of B1 is performed by a single entity, while the *a* action of B2 is the result of the synchronization between two entities, namely the two sub-expressions that form the parallel construct.

We shall extend the action labels with the number of entities, or sub-expressions, involved in performing it. Indeed, these numbers indicate how many action prefixes are "consumed" by the execution of the event.

The standard LOTOS inference rules are modified by associating to every transition label an integer *n* called *synchronization parameter*. An instance of the new, extended transition relation will have the following form:

B1 ---- < x, *n* > ----> B2.

Let SLTS(BP) be the labelled transition system associated to the defining behaviour expression *BP* of a process definition *PD* of process *P*. We shall let *SyncDegree(BP)* denote the maximum of the synchronization parameters occurring in SLTS(BP).

More formally *SyncDegree(BP) = max{ n / B--< x, n >-->B' is a transition in SLTS(BP)}*. The inference rules that define the extended transition relation are extensions of the standard rules: the most relevant case is that of the synchronization rule for the parallel operator. The complete set of extended rules for Basic LOTOS is provided in Appendix B.

CONCLUSIONS

Non-standard interpretations of LOTOS specifications, also called view functions, have been proposed as a most convenient and conservative way to extend the expressivity of the language without affecting its standard syntax and transition-system-based semantics. We envisage two fundamental and related usages of view functions.

1. Some requirements may be formulated in terms of specific view functions f1, ..., fn, and specific views v1, ..., vn: the formal specifier is then required to write a LOTOS specification S is such that f1(s) = v1, ..., fn(s) = vn.

2. In the context of a transformational design and implementation method based on LOTOS, some transformation step S1-->S2 between two specifications may be characterized by the requirements that some (standard) semantic relation between S1 and S2 be preserved (*correctness preservation requirement*), and that some relation between the views f(S1) and f(S2), or some independent requirement on f(S2), be fulfilled, where f is a predefined view function (*transformation requirement*). For instance, one could ask ParDegree(B_{S2}) $\geq$ ParDegree(B_{S1}), in order to increase the degree of parallelism of S1, or SyncDegree(B_{S2}) = 2 for not allowing more then processed to synchronize on any gate.

A number of useful tools could be developed for supporting the computation and manipulation of view functions and views. For those view functions that are defined by structural induction on the syntax of LOTOS, tools for view generation can easily be implemented by means of (meta-)tools based on attributed grammars.

We are now developing a general framework for LOTOS non-standard interpretations based on the denotational approach within an algebraic setting. In such an approach, starting from a grammar which defines the abstract syntax for LOTOS, a *sort set* $\mathfrak{L}$ and a *signature* Λ are derived in such way that the abstract syntax is exactly the term algebra T_Λ, which is initial for any interpretation domain, when such a domain is a Λ-algebra. This means that there exist a *unique* morhism, namely the interpretation function, from the abstract syntax to the interpretation domain. In other words, the "pattern of recursion" of all interpretation functions is the same and depends only on the structure of the abstract syntax terms. An interesting implication of this approach is that such a pattern of recursion can be "frozen" into a polymorphic, higher-order function $\mathfrak{I}$, written in a suitable functional language, which will take abstract syntax terms as well as *all* the operations of *any* actual Λ-algebra as arguments, returning values in that algebra: any interpretation function will be nothing more than the instantiation of $\mathcal{I}$ on the interpretation domain. A preliminary prototype of a transformational environment for LOTOS has been implemented in Miranda[2] [T86] and is described in [L90].

For implementing view functions defined in structured operational style, such as the SyncDegree introduced in the paper, one may most likely take advantage of already existing implementations of the standard dynamic LOTOS semantics.

Some topics for further research are listed below.

- Define further view functions for a better assessment of the range of expressive possibilities offered by the method.

- Differentiate between view functions used as *definitions* of properties and view functions used as *approximations* of possibly undecidable properties.

- Investigate the differences between the denotational and structured operational methods for the definition of view functions, in terms of expressivity.

ACKNOWLEDGEMENT

We wish to thank Rom Langerak, Rudolf Roth, David De Frutos and Yolanda Ortega Mallen for useful discussions that stimulated the development of the ideas presented in the paper.

REFERENCES

[B88] E. Brinksma, "On the Design of Extended LOTOS", Ph.D. Thesis, University of Twente, 1988.

[B89] E. Brinskma, "Specification modules in LOTOS", Proceedings of the 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols - FORTE89, Vancouver, 1989.

[B90] T. Bolognesi, "A Graphical Composition Theorem for Networks of LOTOS Processes", Proceedings of the Tenth International Conference on Distributed Computing Systems (ICDCS-10), Paris, 1990 (IEEE Computer Society).

[BB87] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", Comp. Networks and ISDN Systems, Vol. 14, No 1, 1987.

[BBFN90] T. Bolognesi, P. Boehm, A. Fantechi, E. Najm (eds.), "Correctness Preserving Transformation", document ESPRIT / LotoSphere Lo/WP1/T1.2/N0020, April 1990.

[BHR84] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe, "A Theory of Communicating Sequential Processes", Journal of the ACM, Vol. 31, No. 3, July 1984, pp. 560-599.

2) Miranda is a Trade Mark of Research Software Ltd.

[BLT90] T. Bolognesi, F. Lucidi, S. Trigila, "From Timed Petri Nets to Timed LOTOS", Proceedings of the Tenth International IFIP WG6.1 Symposium on Protocol Specification, Testing and Verification,, Ottawa, June 1990, L. Logrippo, R. L. Probert, H. Ural editors, North-Holland.

[CPJ85] Clack C., Peyton Jones S.L. *Strictness Analysis - a Practical Approach* Proceedings of the second conference on Functional Programming Languages and Computer Architecture - Nancy, France, 1985

[HTZ89] W. U. Hulzen, P. Tilanus, H. Zuidweg, "LOTOS Extended with Clocks", Proceedings of the 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols - FORTE89, Vancouver, 1989.

[ISO89a] ISO - Information Processing Systems - Open Systems Interconnection- "LOTOS- A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", IS 8807, 1989.

[ISO89b] ISO/IEC JTC1/SC21 N4228, "Proposed Draft Addendum to ISO8807:1988 on G-LOTOS", E. Najm editor, 1989.

[L90] D. Latella, *"LOTOS MIRANDUM* - A functional environment for the implementation of view functions for LOTOS" CNUCE internal Report C90-12, May 1990.

[P81] G. D. Plotkin, "A structural approach to operational semantics", Tech. Rep. DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[QAF89] J. Quemada, A. Azcorra, D. De Frutos, "A Timed Calculus for LOTOS", Proceedings of the 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols - FORTE89, Vancouver, 1989.

[S70] D.S. Scott, *Outline of a Mathematical Theory of Computation* Proceedings of the fourth Annual Princeton Conference on Information Sciences and Systems, Princeton University, 1970

[S77] J. E. Stoy, *Denotational Semantics: the Scott-Stratchey Approach*, The MIT Press, 1977.

[T86] D. A. Turner, *An overview of Miranda* SIGPLAN Notices , Dec. 86.

Appendix A - Formal definition of the LOTOS process Functionality parameter

In the following we shall instantiate the definition schema sketched in Section 4 on the *functionality* parameter for Basic LOTOS.

The definition of the interpretation functions makes use of two auxiliary functions defined below:

MinFunc, MaxFunc : $\mathrm{Func}^2 \rightarrow \mathrm{Func}$.

Recall that Func = {0, <>}. For all functionalities *x*

MinFunc 0 x	= 0
MinFunc x 0	= 0
MinFunc <> <>	= <>
MaxFunc <> x	= <>
MaxFunc x <>	= <>
MaxFunc 0 0	= 0

We are now ready for the definition of the *functionality* view function;
FuncInt : ProcDefs → Func

FuncInt [**process** P [g_1,..,g_{1n}] : F := B **where** PD_1,..,PD_k **endproc**] =
BexFunc[B]PDefsFunc[PD_1,..,PD_k]

BexFunc is defined by structural induction on *behaviour expressions* as follows: For all

behaviour-expression	B, B_1, B_2
gate-identifier	g, g_1,..,g_n,.., g'_1,..,g'_m, a_1, ..., a_n
parallel-operator	op
functionalities	ϕ_1,..,ϕ_k

BexFunc[**stop**]ϕ1..ϕ_k	= 0
BexFunc[**exit**]ϕ1..$\phi\kappa$	= <>
BexFunc[i; B]ϕ1..$\phi\kappa$	=BexFunc[B]ϕ1..$\phi\kappa$
BexFunc[g;B]ϕ1..$\phi\kappa$	=BexFunc[B]ϕ1..$\phi\kappa$
BexFunc[**choice** g_1,..,g_n **in**[g'_1,..,g'_m][]B]ϕ1..$\phi\kappa$	= BexFunc[B]ϕ1..$\phi\kappa$
BexFunc[**par** g_1,..,g_n **in**[g'_1,..,g'_m] op B]ϕ1..$\phi\kappa$	= BexFunc[B]ϕ1..$\phi\kappa$
BexFunc[**hide** g_1,..,g_n **in** B]ϕ1..$\phi\kappa$	= BexFunc[B]ϕ1..$\phi\kappa$
BexFunc[(B)]ϕ1..$\phi\kappa$	= BexFunc[B]ϕ1..$\phi\kappa$
BexFunc[B_1[] B_2]ϕ1..$\phi\kappa$	= MaxFunc (BexFunc[B_1]ϕ1..$\phi\kappa$)(BexFunc[B_2]ϕ1..$\phi\kappa$)
BexFunc[B_1[> B_2]ϕ1..$\phi\kappa$	= MaxFunc (BexFunc[B_1]ϕ1..$\phi\kappa$)(BexFunc[B_2]ϕ1..$\phi\kappa$)
BexFunc[B_1op B_2]ϕ1..$\phi\kappa$	= MinFunc (BexFunc[B_1]ϕ1..$\phi\kappa$)(BexFunc[B_2]ϕ1..$\phi\kappa$)
BexFunc[B_1>> B_2]ϕ1..$\phi\kappa$	= MinFunc (BexFunc[B_1]ϕ1..$\phi\kappa$)(BexFunc[B_2]ϕ1..$\phi\kappa$)
BexFunc[P_i [a_1, ..., a_n]]ϕ1..$\phi\kappa$	= ϕ_i

The functionality of a process instantiation is defined to be equal to the *interpretation* of its related process definition. Here is the main difference with [ISO89] where the functionality of a process instantiation is defined simply as that (syntactically !) declared by the user in the header of the process definition, so that process definitions like that given in Section 4 for process *P* are considered *correct* w.r.t. static semantics. Under our definition of the functionality view, at least something like a *bad quality* warning should be issued for such process definitions. It is also worth comparing our definition for the *enabling* operator >> with the standard one where *func(B_1>> B_2) = func(B_2)* so that *func(B_1>> B_2)* may be equal to <> also when B_1 definitely fails to successfully terminate, namely *func(B_1) =0.*

Finally, the definition of *PDeclFunc* is:

PDefsFunc[**process** R_1 [g_{11}, ..., g_{1n_1}]: F_1:= B_1 **endproc**

...

process R_k [g_{k1}, ..., g_{kn_k}]: F_k:= B_k **endproc**] =

$\mu(\phi 1..\phi\kappa).<\text{BexFunc}[B_1]\phi 1..\phi\kappa, ..., \text{BexFunc}[B_k]\phi 1..\phi\kappa>$

Continuity of *BexFunc[B]* can easily be proved (see [L90]) by structural induction on *B* when *Func* is made a c.p.o. by means of $\ll_\phi \subseteq \text{Func} \times \text{Func}$ defined as follows:

$$(\forall x,y \in \text{Func})[x \ll_\phi y \text{ iff } (x = 0 \vee x=y)]$$

One has simply to observe that all chains in *Func* are finite and that *MinFunc* and *MaxFunc* are monotonic. Safety of *BexFunc[B]* can also be proved proved [L90] by structural induction on *B*.

Appendix B - Formal definition of SyncDegree

Extended Inference Rules of Transition

Let:		
	g, g1,..,gn	range over the set G of *user-definable* gates;
	i	denote the unoservable action;
	μ	range over Act = G ∪ {**i**};
	δ	denote the successful termination action;
	g^+	range over $G^+ = G \cup \{\delta\}$
	μ^+	range over $Act^+ = Act \cup \{\delta\}$

exit)	**exit** ---- <δ, 1 > ----> **stop**	is an axiom.
act prefix)	μ ; B ---- < μ, 1 > ----> B'	is an axiom.
sum)	(B)[gi / g] -- <μ^+, n> --> B'	implies
	choice g **in** [g_1,..,g_n]B -- <μ^+, n> --> B'	
par)	Let op be a parallel operator	
	(B)[g1 / g] op.. op(B)[gn / g]-- <μ^+, n> --> B'	implies
	par g **in** [g_1,..,g_n] op B -- <μ^+, n> --> B'	
parenthesis)	B-- <μ^+, n> --> B'	implies
	(B) -- <μ^+, n> --> B'	
choice)	B1 -- <μ^+, n> --> B1'	implies
	B1 [] B2 -- <μ^+, n> --> B1'	
	B2 -- <$\not{c}^+$, n> --> B2'	implies
	B1 [] B2 -- <μ^+, n> --> B2'	

parallel) B1 -- <μ, n> --> B1', $\mu \notin$ [g1, ..,gn] implies
B1 |[g1, ..,gn]| B2 -- <μ, n> --> B1' |[g1, ..,gn]| B2

B2 -- <μ, n> --> B2', $\mu \notin$ [g1, ..,gn] implies
B1 |[g1, ..,gn]| B2 -- <μ, n> --> B1 |[g1, ..,gn]| B2'

B1 -- <g^+, n> --> B1', B2 -- <g^+, m> --> B2', $g^+ \in S \cup \{\delta\}$
implies
B1 |[g1, ..,gn]| B2 -- <g^+, m + n> --> B1' |[g1, ..,gn]| B2'

hide) B -- <μ^+, n> --> B', $\mu^+ \notin$ g1, ..,gn implies
hide g1, ..,gn **in** B -- <μ^+, n> --> B'

B -- <g, n> --> B', g $\in$ g1, ..,gn implies
hide g1, ..,gn **in** B -- <i, n> --> B'

proc inst) Let B = **process** P[g1', ..,gn'] := Bp **endproc** be a process definition,

Bp [g1/g1', ..., gn/gn'] -- <μ^+, n> --> B' implies
P[g1, ..,gn] -- <μ^+, n> --> B'

enabling) B1 -- <μ, n> --> B1' implies
B1 >> B2 -- <μ, n> --> B1' >> B2

B1 -- <δ, n> --> B1' implies
B1 >> B2 -- <i, n> --> B2

disabling) B1 -- <μ, n> --> B1' implies
B1 [> B2 -- <μ, n> --> B1' [> B2

B1 -- <δ, n> --> B1' implies
B1 [> B2 -- <δ, n> --> B1'

B2 -- <μ^+, n> --> B2' implies
B1 [> B2 -- <μ^+, n> --> B2'