Richard O. Sinnott and Kenneth J. Turner. The Specification of A Type Management System in LOTOS. Technical Report CSM-122, Department of Computing Science and Mathematics, University of Stirling, Scotland, April 1994.

The Specification of a Type Management System in LOTOS

Richard Sinnott and Prof K.J. Turner,
Department of Computing Science,
University of Stirling,
Stirling FK9 4LA,
Scotland
email: ros || kjt @cs.stir.ac.uk

Abstract

This paper offers one approach to developing a type management system in LOTOS [4]. The limitations of the definitions of type and subtype, as given in the Reference Model for Open Distributed Processing (RM-ODP) [1], when used to model a type management system in LOTOS are investigated and different levels of type equivalence identified. An approach is then given to modelling types in LOTOS whereby type relationships can be established that are not based on simple name equivalence only and these are then used in the development of a prototype type management system.

1 Introduction

In the development of a distributed system, it has been identified that there exists the need for a common understanding of the nature (type) of information, regardless of its representation in a particular system [3]. To this end a *type manager* has been identified as an entity that is expected to fulfill this role, *i.e.* it should provide the functionality to describe and compare types, as well as to govern the names of types and the relationships that exist between them.

The aim of this paper is to give an account of the approach to formally specifying in LOTOS a basic type manager that will allow for type checking and comparison between types. The term "basic" is used here to represent the fact that the type manager specified here does not deal with types that have unknown representations. In fact the type manager deals here only with types that are globally known, *i.e.* known throughout the specification. It should be pointed out that this is not a limitation of this particular specification; it will always be the case that any LOTOS specification can only deal with "types" whose name and representation are defined in the specification. This work does, however, address some of the issues involved in realising distributed type management.

This paper draws on the work of Part 2 of the RM-ODP, [1], especially in its definitions of types and subtypes [1]. The limitations of these definitions when used to develop a type management system in LOTOS are given and different forms of type equivalence are identified which enable issues involved in federating type managers to be considered.

The rest of this paper is structured as follows: section 2 introduces definitions of types and subtypes as given in [1] and the problems involved in attempting to formalise these definitions in the development of a type management system in LOTOS; section 3 introduces a new type modelling approach in LOTOS which is more applicable to a type manager; section 4 gives an

[1] Throughout this paper the term subtyping implies a supertyping relation exists also.

overview of a prototype type manager specification using the type modelling approach developed in section 3; section 5 focuses on issues that have to be addressed when federating type management systems and finally section 6 draws some conclusions and identifies future extensions to this work.

2 Types and Subtypes

2.1 Types and Subtypes in ODP

Part 2 of the RM-ODP [1] states that a type is simply a predicate characterising a collection of <**X**>s. An <**X**> is of the type, or satisfies the type, if the predicate holds for that <**X**>. Here <**X**> may be an object, interface or action.

As may be seen, this is a very general notion of type and one which proves to be problematic when interpreted in LOTOS. It should be noted, however, that it is generally difficult to give a formal definition of the notion of type, and not only the ODP definition which is problematic.

The notion of subtyping in ODP is directly linked to the notion of type. The RM-ODP (Part 2) [1] gives the subtyping (and supertyping) relationship as "a type A is a subtype of a type B, and B is a supertype of A, if every <**X**> which satisfies A also satisfies B". Once again we have a very general notion which is problematic when interpreted in the FDT LOTOS.

2.2 Types and Subtypes in LOTOS

Following from the very general notion of type as given above, numerous problems arise when this notion is applied to LOTOS [4]. The main problem stems from the generality of the type concept. Firstly, the characterising predicate can be associated with different things (*e.g.* an object, interface or action) and secondly, it does not state which predicates a given type must satisfy. Thus, how is it possible to specify a "type" in as general a way as the ODP definition? In short the answer is, it cannot be done in LOTOS. LOTOS allows predicates to be written in many different ways. However, a general predicate that applies to all notions of type is not possible. In LOTOS one specifies something explicitly, whereas the notion of type as given here is at a rather "hand-waving" level. The best that LOTOS can manage is to give explicit examples of types and their relationships. That is, we are limited in LOTOS in that we have to have a problem domain. This notion of type is so general it has no explicit problem domain. Goguen and Wolfram [8] also identify the complications and limitations of systems using the notion of "type" in such a general manner.

The question now arises as to how a non-general notion of type can be represented in LOTOS. What kind of predicates can be given and how can these predicates be represented in LOTOS? The determining factor here is that the representation of types must allow for a means of type checking. That is, we know that the type manager has to manage types of information and the inter-relations between these types. Thus from the type manager's point of view, a question that might be asked is, given a type what are its subtypes? Are two types in a relationship (*e.g.* is one type a subtype of another)? Are these two types equivalent? Therefore it is also necessary to consider how subtyping may be achieved in LOTOS. This should be in such a way that comparisons between types can be made and consistency checks facilitated.

Within LOTOS the ways in which types and subtyping may be represented directly are through the process algebra part or ACT ONE. Both of these have limitations, however, in providing a type system suitable for use by a type manager, as will be shown.

2.3 The Process Definition Approach

The modelling of a type through the process definition approach is the most obvious course with regard to the current architectural semantics work [2]. However, in this situation it is not particularly useful. Part 4 of the RM-ODP [2] does not explicitly relate the notion of a type to LOTOS. As stated previously, this is due to the fact that the notion is such a general one. Part 4 does, however, attempt to explain how subtyping and supertyping may be modelled between template types, where a template type may be regarded as a predicate defined as the ordered pair (*template, instance of*). Thus object X is of type (*template, instance of*) implies that object X is an *instance of template*, where *template* is a process definition and an *instance* of the template is defined as an instantiation of the template or an object that is an acceptable substitute for an instantiation of that template. This acceptibility relation depends upon the context in which the object is to be used. It might be the case that the acceptability relation is one based on behaviour compatibility of the complete behaviour, *i.e.* the instantiations of the process definitions — this would be the strongest acceptability relation. Alternatively, the acceptability relation might be based on a certain aspect of behaviour, *e.g.* has some event which accepts a natural number.

Although this method of defining types and subtypes is valid it is not particularly useful as far as the type manager is concerned. It is not possible using this type/subtype model to return the subtypes of a given type, or list the properties of a type, or even to check that two types are in a subtyping relation within the constraints of LOTOS. Consider the following fragment of LOTOS where a subtyping relation is modelled through extension [2] between two template types, *i.e.* two process definitions and their associated instantiations.

```
              Temp1[g1,g2] ||| Temp2[g2]
where
 process Temp1[g1,g2]:noexit:=        process Temp2[g2]:exit(Nat):=
  (g1 ?n:Nat[n eq 0]; stop              g2 ?n: Nat[n ne 0];
  []                                    exit(n)
  Temp2[g2])                          endproc (* Temp2 *)
  >> accept newn: Nat in
     g1 !newn; Temp1[g1,g2]
 endproc (* Temp1 *)
```

Here the template type of process definition *Temp1* and its instantiation also satisfies the predicate associated with the template type of process definition *Temp2* and its instantiation. This is because process definition *Temp2* makes up one part of the behaviour expression of *Temp1* and the instantiations of both processes are compatible. That is, the part of the behaviour expressions that *Temp1* and *Temp2* have in common are both instantiated with gate *g2*. Thus in an environment expecting *Temp2* instantiated with gate *g2*, an instantiation of *Temp1* with gate *g2* as the second gate, would suffice.

As stated, the specification cannot derive any relationship between these two template types. Establishing type relationships here may only be achieved outside of the specification through some form of conformance or behaviour compatibility relation.

Another problem with this approach is that the specification of the type manager is not scalable. Consider the addition of a new type to the type manager. If the above approach is taken then the type manager must have the above type template contained within itself waiting to be instantiated. That is, one might expect the type manager to accept a message of the form *add_type !request !user_id !some_type*. What is this "*some type*"? If a type is given as a process

[2] Extension may be regarded as the simplest and most easily understood method of achieving subtyping.

definition then this is not a valid LOTOS expression. The alternative is that it is some identifier (role) for a behaviour which will be instantiated from the type manager which is itself a template (process definition). This means however that the type manager must know all possible types (process behaviours) before requests are received which is certainly not a scalable solution.

An alternative to modelling types and subtypes as instantiations of process definitions is to model them through ADTs in the ACT ONE part of LOTOS.

2.4 The Abstract Data Type Approach

The ADT approach is to model types as instances of sorts in the ACT ONE part of LOTOS. Consider the ADT *X_type* given here:

```
type X_type is
     sorts X
     opns a: -> X
          b: X -> X
          c: X,X -> X
endtype (* X_type *)
```

```
type Y_type is
     sorts X, Y
     opns a: -> X
          b: X -> X
          c: X,X -> X
          d: X -> Y
endtype (* Y_type *)
```

In the behaviour part of LOTOS, when a value is declared of sort X then its sort defines the predicate. For example, if we have *x: X* declared then the predicate satisfied by x is that it has the operations *a,b* and *c* and satisfies the equations (if any are given) [3].

The problem with this approach as far as the type manager is concerned, however, is that it is not possible to establish subtype relationships or type equivalence in a form that is checkable, *i.e.* in a form not based entirely on name equivalence. Before going on to state why this is the case it is necessary first to establish formally what exactly is meant by type equivalence and subtyping in ADTs.

Type equality may be interpreted in terms of subtyping. Thus if $\leq$ is given as the subtyping relation between two ADTs *adt1* and *adt2* say, then *adt1* equals *adt2* if:

$$adt1 \leq adt2 \wedge adt2 \leq adt1$$

that is, they are both subtypes of each other. Given two ADT signatures *adt1* and *adt2*, then *adt1* is a subtype of *adt2* provided the operations of *adt2* are contained in *adt1*, that is, *adt1* has all of the operations of *adt2* and possibly more. For any ADTs for which subtyping is expected to hold this implies [4]:

1. all operation names in *adt1* have corresponding operation names in *adt2* (corresponding in the sense of representing the same operation);

2. the common operations have the same number and sorts of input parameters (in the same order) (arguments);

3. the common operations have the same prefix/infix nature;

4. the common operations produce the same result sorts [5] (terminations).

[3] If no equations are given then we have only a signature type, as opposed to full behavioural subtyping.

[4] As identified in [9], this is a restrictive subtyping relation in that all of the operations of the supertype have to be included in the subtype, even if they will not be invoked in a given context.

[5] There is no need to consider the number of result types, as ACT ONE operations only return one result.

Consider now the ADT *Y_type* given above. From the type rules given above, one would expect that *Y_type* was a (signature) subtype of *X type*. However, LOTOS establishes types through instances of sorts, *e.g. y: Y* and *x: X*. The type constructs of LOTOS merely provide a means to collect sorts and operations (and equations). It is not the case here that *y: Y* is a (signature) subtype of *x: X*, as the corresponding operations (as defined in the types rules) in the signature of *Y_type* are based on *X* and not *Y*. Thus when *y: Y* is declared in the process algebra part of LOTOS, it is not valid to perform operations *a,b* and *c* on it. These may only be performed on *X* sorts. Thus it can be seen that LOTOS uses name equivalencing when type checking is done.

As far as the type manager is concerned and its specification in LOTOS, this is a serious restriction on the usefulness of ADTs to specify "types" directly, as it limits the checking that a type manager can perform. A type manager which uses only name equivalence is in effect a type manager based upon "trust". That is, the type manager trusts that the type names are associated with the correct types and does not check otherwise. Ideally a type manager should be able to identify structural similarities between ADT signatures as well as name equivalence, where structural similarity is defined similarly to subtyping of ADT signatures with looser constraints on the inputs and result parameters. Specifically; the rules of the subtyping rules given above should be modified to:

2. the common operations have the same number and sorts of input parameters (in the same order) and these must be used consistently throughout the entire signature;

4. the common operations produce the same result sorts and these must be used consistently throughout the entire signature;

This consistent usage of a sort in a signature means that two ADTs using different sorts (labels) are structurally similar only if the sorts in the ADTs signatures can be relabelled consistently throughout the signature so that the corresponding operations they share are identical. If the corresponding operations they share represent the whole signatures of the ADTs, then the structural similarity the ADTs share is in fact a structural equivalence. Another way of considering this is that if one ADT is structurally similar to a second ADT, and the second ADT is structurally similar to the first, then they are structurally equivalent. Thus in effect, structural equivalence corresponds to type equivalence, and structural similarity corresponds to subtyping provided it can be established by some means (see 5.1) that the sorts in the signatures are semantically equivalent. Consider the following ADT signatures:

```
type A_type is
     sorts A
     opns a: -> A
          b: A -> A
          c: A,A -> A
endtype (* A_type *)
```

```
type B_type is
     sorts B
     opns a: -> B
          b: B,B -> B
          c: B,B -> B
endtype (* B_type *)
```

```
type C_type is
     sorts C
     opns a: -> C
          b: C -> C
          c: C,C -> C
          d: C,C,C -> C
endtype (* C_type *)
```

Here, *A_type* is structurally equivalent to *X type* given above, as relabelling the *A* label with an *X* label makes the *A_type* signature identical to the *X type* signature. *X type* is structurally similar to *C_type* as relabelling the *C* label (sort) by an *X* label in the *C type* signature makes the corresponding operations they share (*a,b,c*) identical. However, *X type* has no immediately identifiable structural relationship to *B type* as operation *b* requires too many input parameters, hence relabelling will not satisfy the above rules for structural similarity.

In fact in this example there exists a direct isomorphism between ADTs *A type* and *X type*. That is, they differ only in the label that is attached to the carrier in the signature, *i.e.* the

sort. Thus a bijection exists from *A type* to *X type* (and hence an inverse relation) which is achieved by relabelling the sort (X or A) in the two ADT signatures. It is worth noting that in homogeneous algebras, *i.e.* algebras with a single sort as given here in *A type*, rule 2 of the subtyping rules requires that only the number of parameters of corresponding operations is the same as relabelling will always be consistent. In heterogeneous algebras, *i.e.* algebras with more than a single sort as in *Y type* given above, the number and sorts of parameters and results need to be considered for corresponding operations. Thus for example, given the following ADTs:

```
type P_type is              type R_type is
     sorts P, Q                  sorts R, S
     opns a: -> P                opns a: -> R
          b: P -> P                   b: R -> R
          c: P,P -> P                 c: R,R -> R
          d: P -> Q                   d: S -> R
endtype  (* P_type *)      endtype (* R_type *)
```

P_type is structurally equivalent to *Y type* above as relabelling X and Y in the signature of the ADT *Y type* by labels P and Q respectively would give the same signature as ADT *P type*. However, this is not the case with *Y type* and *R type*, *e.g.* relabelling X and Y by R and S respectively would result in operation d having R as input and S as output, which is not the same as ADT *R type*.

Following from this discussion, a notion of type equivalence may be introduced at three main levels. These are:

1. name equivalence;
2. structural similarity;
3. name and structural similarity;

As stated previously, LOTOS only permits name equivalencing in ACT ONE. On its own this represents a rather limited type checking policy for a type manager. It does, however, enable type equivalences to be given directly.

The ability to be able to identify structural similarity and possibly structural equivalence between types is a desirable property for a type manager to possess, as it offers the possibility for a stronger form of type checking, *i.e.* it does not assume (trust) that a type name is necessarily associated with a certain structure. Being able to identify structural similarity also enables browsing of types to be facilitated. For example, requests of the form, "I know what structure (or partial structure) for a type I want but I do not know the name", could be satisfied.

Being able to provide name and structural similarity checking (and possibly structural equivalence checking) represents full signature type checking. Here the type manager checks that a given type has the correct name and structure associated with that type. This represents the strongest of the three forms of type checking given here.

2.5 The Failure of the Direct ADT Approach

Having identified the different forms of type checking possible and noted that establishing structural equivalence is a desirable property for a type manager to possess, the question now arises as to how can structural equivalence be established in LOTOS? That is, how can it be established that *x: X* is structurally equivalent to *a: A*? The specifier can see immediately that this is the case, but the specification cannot derive this fact for itself. To establish structural equivalence would require something of the form:

```
all_ops: X -> Op_Set
```

which would return the operation set $\{a, b, c\}$ (with the associated input and output parameters for each operation) for X and for A. Once it is known that the operations on these sorts have the same names, number of inputs and differ only in the label attached to the sorts throughout the signature then structural equivalence could be checked by an operation of the form:

```
is_se(X,A) = all_ops(relabel[X/A]) eq all_ops(A)
```

where *is_se* checks for structural equivalence, *relabel* takes two parameters and replaces all occurences of the first parameter by the second in a given set of operations, and *eq* checks equality of two sets. The problem is, however, LOTOS cannot have an operation like *all ops*. That is, it cannot return a set of the operations associated with a given sort. Thus this approach is also not applicable to the development of a type manager.

3 The ADT Describing ADT Approach

From the above arguments it became obvious that the first two methods in LOTOS of representing types and subtypes were both unsuitable in the development of a type manager. The solution that was adopted came from considering what was lacking in the ADT approach. The simplistic idea of subtyping as a comparison between instances of sorts and their operations (and equations) was lacking only in a means to compare instances.

The solution found was to model types and subtype relations as ACT ONE describing ACT ONE[6]. That is, it is necessary to model the operation *all ops* to enable structural equivalence relationships between types to be checked. Thus when given types like *X type* and *A type* given above we would like to be able to determine whether they are structurally equivalent to one another. This requires checking that they possess corresponding operations. In LOTOS an operation may be given by the quadruple:

```
< Operation_Name, In_Pre_fix, Input_Sorts, Output_Sort >
```

where Operation_Name is simply an identifier (not necessarily unique), In_Pre_fix is an identifier denoting whether the operation is prefix or infix, the Input_Sorts are a list of sort names (possibly an empty list as in operation a from ADT X above), and the Output_Sort is a sort name.

To compare whether two operations are the same in LOTOS it is necessary to check that they have compatible operation quadruples, *i.e.* same names, same prefix/infix notation, and compatible inputs and results. Comparison of operation names is trivial in LOTOS as it reduces to comparison of identifiers. Checking the infix or prefix nature of operations may be achieved simply by ensuring that when an operation is defined in LOTOS, a label is given denoting the operation as prefix or infix. Checking compatible input and output sorts may be achieved by comparison of identifiers.

Using this approach of breaking down an operation into separate components, it is necessary to have some method of putting the components together again in a form that will be understood by LOTOS. The approach adopted uses the operation *make op*. This is given by:

```
make_op : Op_Name, In_Pre_fix, Sort_List, Sort -> Op
```

[6]This method for generating typing and subtyping although valid is metaphysical in that a language is being used to describe itself.

where *Op_Name* is simply an identifier, *In_Pre fix* is an identifier denoting the infix/prefix nature of the operation, *Sort_List* is a list of identifiers, *Sort* is an identifier, and *Op* represents the operation. Thus equality of operations may be checked by:

```
op1 eq op2 = ((get_name(op1) eq get_name(op2)) and
              (get_inpre(op1) eq get_inpre(op2)) and
              (is_eq_list(get_inputs(op1),get_inputs(op2))) and
              (is_eq_sort(get_output(op1),get_output(op2))));
```

where

```
get_name(make_op(opn1,inpre,sl1,s2)) = opn1;
get_inpre(make_op(opn1,inpre,sl1,s2)) = inpre;
get_inputs(make_op(opn1,inpre,sl1,s2)) = sl1;
get_output(make_op(opn1,inpre,sl1,s2)) = s2;
```

and operation *is_eq_list* checks whether two lists are the same and returns a Boolean result, *is_eq_sort* checks whether two sorts names are the same and returns a Boolean result. Having a method of determining whether two operations are the same reduces the problem of (signature) type equivalence (and subtyping) between ADTs to a trivial set comparison, where set elements are the created operations. Thus subtyping is given by subsetting of operations and type equivalence is given by two sets of operations containing identical members.

With this approach, it is necessary to create types explicitly. Thus if ADT *X type* given above were to be created, this would be given by:

```
sort(X) = Insert(X,{})
ops(X)  = Insert(make_op(a,prefix,<>,X)),
          Insert(make_op(b,prefix,add_sort(X,<>),X),
          Insert(make_op(c,prefix,add_sort(X,add_sort(X,<>)), X), {})))
```

where <> gives an empty *Sort List*, {} gives empty sets to contain the operations and sorts of the signature, *add_sort* inserts an identifier into a list, *prefix* declares the operation to be of prefix nature and *Insert* adds an element to a set. Thus subtyping between two types, *y: Y type* and *x: X_type* can be checked by an operation as shown here:

```
sub(y,x) = ops(y) Includes ops(x);
```

where *Includes* checks whether one set is a subset of another set.

As stated previously, this approach permits signature checking of types. It might also be possible to apply the same approach to the equations of the ADTs though. That is, they could be reduced to a string compatibility check: however, this would be even more metaphysical in that the semantics (one way of describing the equations of ADTs) of the operations would be reduced to a syntactical level. There might also be many different ways to represent the same equations in LOTOS and notions such as congruence will have to come into effect. Hence the equation part of ADTs has been left in this work so far.

4 The Structure of the Type Manager Specification

4.1 Description

The type manager specification has the structure as shown in Figure 1, where the shaded region represents hidden gates (*db type* and *access check*) and the gates *add t,_del t,_list t,_check rel* and *all_subs* parameterise the specification.

It should be noted that the approach taken was deliberately simplistic. A simple global hierarchical type structure was specified in ACT ONE using the method shown above in section 3. This was then used for testing the functionality of the type manager.

The type manager specification consists of two main processes, *Process Type Admin* and *Process Authority Check*. *Process Tester* is there primarily as a test harness process for testing the specification using the LOTOSPHERE LITE Tool [5].

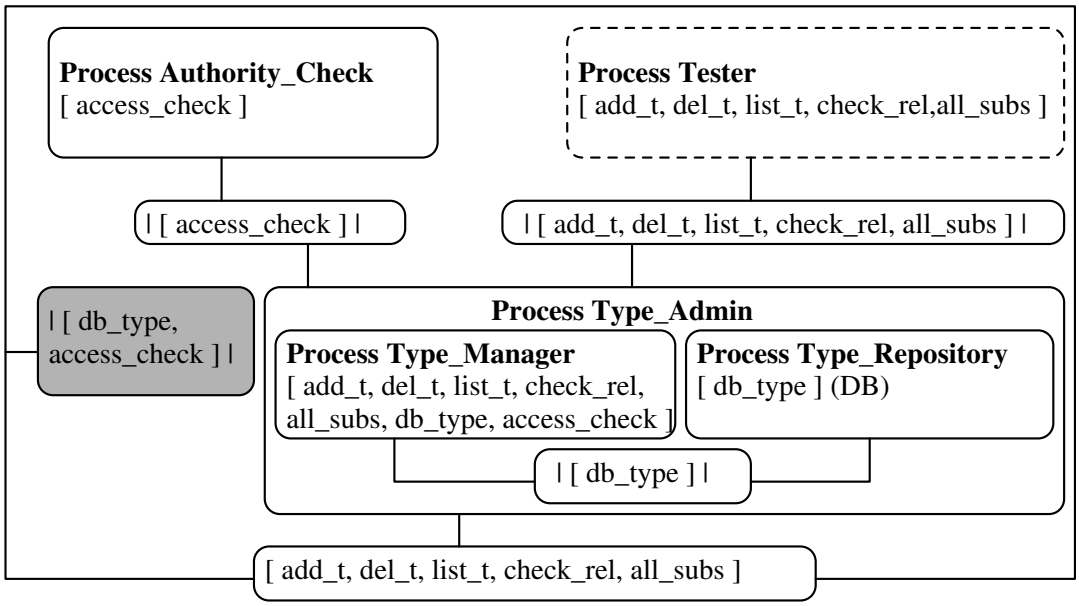

Figure 1: The Overall Type Manager Structure

Process Type Admin acts both as the interface to the environment of the type manager (*Process Type Manager*) and as the type information storage system (*Process Type Repository*). The environment is provided by *Process Tester* [7], that is, this process models possible user behaviours. *Process Authority Check* is there to provide a security function. When *Process Type Admin* (*Process Type Manager*) receives a request from the environment (from *Process Tester*) via synchronisation at a given gate from *add t, del t, list t, check rel* or *all subs* it first synchronises with *Process Authority Check* at the hidden gate *access check*. This process determines whether or not the user has enough authority to perform the requested operation. Depending upon the ODP security function (which has not been specified here) either an appropriate error message is returned (via synchronisation at the same gate as the initial request was received from), or *Process Type Manager* synchronises on the hidden *db type* gate with *Process Type Repository*. This process is initially given an empty database (store) in which information about the types is to be kept. Upon receipt of a request from *Process Type Manager*, this process then checks the details of the user request. This will result in an appropriate error message being returned, modification to the database taking place, or the required information being returned, depending upon the user request. The appropriate response is communicated back to *Process Type Manager* at gate *db type*. This process then returns the result to the user via synchronisation at the gate from which the initial request was received.

The structure of the specification is deliberately simple, with the processing of all operations following similar forms. Notions of access checking have been largely avoided although this will be an extremely important part of a type manager in a distributed system.

[7] This is shown as a dotted line as it is not really part of the specification. It is used only for testing purposes.

The set of operations specified for the type manager represents the minimum functionality that a given type manager is expected to have. These operations include:

- **Add_Type** — add a new type to the type manager repository;
- **Del_Type** — delete an existing type from the type manager repository;
- **List_Type** — list the properties associated with a particular type;
- **Check_Rel** — check whether two types are in a given relationship;
- **All_Subs** — return all the subtypes of a given type.

Additional functionality for the type manager would be trivial to specify, *e.g.* operations to add or delete type relationships.

5 Additional Type Management Considerations

5.1 Dictionary of Type Names

The treatment of type equivalence so far has dealt purely at a syntactic structuring level only. There is no real notion of true type equivalence as such. Rather, the ADTs represent an empty shell with no semantics other than that suggested by the operation names.

To be really qualified as a type management system, a type manager should deal with the behaviour (semantics) of types also. Thus although X *type* and A *type* given above are type equivalent up to isomorphism, their semantics might be entirely different. For example, label X might represent natural numbers and label A Booleans with operations *a,b,c* representing "0", "succ" and "+" for X, and "true", "not", "or" for A respectively. Therefore they can only be considered type equivalent if the sorts used in the signatures can be established as being the same semantically.

Establishing complete behavioural equivalence of types is an active area of research. A type manager should ideally deal with the semantics of types. One possibility for the type manager to avoid the problems of establishing full type equality itself (*i.e.* including type semantics) is through a naming approach, *i.e.* when a new type is supplied to the type manager, its name is associated with semantics. The semantics might well be supplied by human intervention. The problem of determining when two types are equivalent then becomes a naming issue.

This raises several issues which have to be addressed. For example, considerable care is required to ensure that the types registered with the type manager are unambiguous and have the semantics as expected. This might be solved through a practical approach. For instance, by restricting the users wishing to access and modify the type manager's repository contents: adding new types, deleting types and modifying existing types.

5.2 Merging Distributed Type Systems

As identified above, much of the difficulty involved in full semantical type management can be alleviated through a type naming approach. This is not without its own difficulties, however, especially as a type manager will be dealing with distributed systems. For example, it is likely that in a distributed system several dictionaries (naming schemes) would exist. A client using one naming scheme wishing to access a server in another naming scheme requires some form of guarantee that the type it requires is the one provided. That is, they might have the same name but a guarantee of the semantics is required also. Alternatively, they might have different names

but semantically they might be the same. This requires that the naming schemes are scalable and also that they can be made compatible.

Being able to establish that two types are structurally equivalent eases the problem involved in merging (and possibly automating) type naming schemes. That is, being able to state that two types are structurally equivalent might not necessarily imply that they are type equivalent, however, types that are not structurally equivalent cannot be type equivalent. Hence establishing structural equivalence enables a restricted set of type names known to the type managers to be given, as opposed to the full set of known type names.

5.3 Relabelling Algorithm Considerations

In order to check for structural equivalence between types when merging type managers, an efficient relabelling algorithm is essential. As shown above with *P type* and *R type*, establishing structural equivalence requires that consistent relabelling of the sorts in the signature is made. In very complex ADTs with numerous sorts many different relabelling possibilities might exist. For two ADTs consisting of n sorts and m sorts respectively, permutation theory gives

$$\frac{n!}{(n-m)!}$$

relabelling permutations. Thus two ADTs which each have five different sorts in their signatures say, offer 120 relabelling possibilities. Therefore to reduce this complexity several basic universally known types should be provided. For instance, Booleans, natural numbers and integers etc represent types which should be universally understood. These may then be removed from the set of sorts contained in the relabelling permutation set. Other types used in ADTs which are not universally known should be labelled explicitly as locally defined, application-specific types.

To further increase the efficiency of the relabelling algorithm, other time saving features can be included when checking for structural equivalence. For instance, before relabelling check that the two ADTs under consideration have the same number of sorts in their signature and the same number of operations. If the ADTs do not satisfy these considerations then it can be stated that they are not structurally equivalent, otherwise the relabelling algorithm must take place to check for structural equivalence.

The exact form of the relabelling algorithm has not yet been defined in this work so far. Possible solutions lie in the direction of graph theory and constructing adjacency matrices. That is, a given ADT signature can be represented by a directed graph with nodes represented by sorts and arcs representing operations.

In this work, all names to semantic mappings are accounted for. Hence no relabelling algorithm has been applied here. Thus establishing name and structural equivalence here corresponds to type equivalence.

6 Conclusions and Extensions to this Work

This work has identified some problems in the notions of types and subtypes as given in [1] when applied to develop type systems in LOTOS. The definitions given in [1] have been identified as very general and as such are not possible to model in LOTOS directly. An approach has been given, however, which enables types and type relationships to be established in LOTOS. This has then been used to develop a prototype type management specification in LOTOS, a copy of which may be obtained by contacting the author in the first instance.

The specification itself has been fully tested with LITE [5], and includes a testing process to aid in the simulation of the specification.

The approach taken in the specification has been deliberately simplistic. A simple hierarchical type structure was specified in the ADTs and used for testing the type manager. It is likely that this simple type hierarchy will be replaced by a more useful (and complex) type hierarchy, *e.g.* a type system specified in the interface definition language CORBA IDL ([7]). This newly modified type hierarchy may then be used in trading and federated trading, where services types and their associated interface types need to be maintained and understood. It is likely that this approach would then compliment other works too. For example, the Service Request Description Language of Popien/Meyer [10] enables structured specification of service requests to be made. The approach taken there is to denote service types as identifiers, hence the approach given here of establishing type relationships between types (identifiers) would enable type checking to be achieved between these services (type identifiers).

There are numerous extensions that can be made to the type management system specified so far. For instance: enabling modification of types; enabling type relationships to be checked when the corresponding operations that types possess have different names; providing a scalable type manager type naming and storage solution to enable faster searches and browsing facilities for types — in the current specification types are simply added to a database with the user providing the type name; enabling type relationships to be established when not all of the operations of the supertype are included in the subtype, *i.e.* allowing the context of the type to influence type relationships; issues of covariance and contravariance can also be achieved with this approach but has been left out of this paper to enable the approach to be given as simply as possible. All of these extensions represent variations on a theme however. That is, the approach given here of enabling signature subtyping to be established in LOTOS is the main result of this paper and the extensions to the work are trivial to achieve.

References

[1] Information Processing Systems - Open Distributed Processing - Basic Reference Model of ODP - Part 2: [Recommendation **X.902 - ISO 10746-2**] *Descriptive Model.*

[2] Information Processing Systems - Open Distributed Processing - Basic Reference Model of ODP - Part 4: [Recommendation **X.904 - ISO 10746-4**] *Architectural Semantics.*

[3] ISO/IEC JTC1/SC21/WG7 N807 - Working Document on Topic 9.1 (ODP Trader).

[4] ISO: *Information Processing Systems - Open Systems Interconnection - LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, International Organisation for Standardisation, ISO 8807, Geneva, August 1989.

[5] M. Caneve and E. Salvatori: *LOTOSPHERE Lite User Manual*, ESPRIT Ref: 2304, Lo/WP2/N0034/V04, LOTOSPHERE Consortium, 1991.

[6] J. Indulska, M. Bearman, K. Raymond: *A Type Management System for An ODP Trader*, Open Distributed Processing 1993, Participants Proceedings, editors J. de Meer, E. Mahr, O. Spaniol.

[7] The Common Object Request Broker Architecture Interface Definition Language, Object Management Group, Document Number 91.12.1.

[8] J. A. Goguen, D. Wolfram: *On Types and FOOPS*, Prog IFIP TC-2 Conference on Object Oriented Databases.

[9] G. Blair, J. Gallagher, D. Hutchison and D. Shepherd: *Object-Oriented Languages, Systems and Applications*, Pitman Publishing, 1991.

[10] C. Popien and B. Meyer, *A Service Request Description Language*, Participants Proceedings, Seventh International Conference on Formal Description Techniques, Berne, Switzerland, 4-7 October 1994, Editors D. Hogrefe and S. Leue.