On Behavioural Subtyping in LOTOS *

H. Bowman[1], *C. Briscoe-Smith*[1], *J. Derrick*[1] *and B. Strulo*[2]
[1]*Computing Lab., Univ. of Kent, Canterbury, Kent, CT2 7NF, UK*
[2]*BT Laboratories, Martlesham Heath, Ipswich, IP5 7RE, UK*
Email: {H.Bowman,cpb4,J.Derrick}@ukc.ac.uk & bstrulo@srd.bt.co.uk

Abstract

We consider how the OO notion of subtyping relates to LOTOS testing theory. In particular, we investigate which of the standard LOTOS preorders is a suitable instantiation of behavioural subtyping and argue that each of the main preorders, trace preorder, trace extension, reduction and extension, is in some way deficient. Then, in the light of pre and post condition based models of OO subtyping, we re-work the basic interpretation applied to LOTOS behaviour descriptions. We argue that this re-interpretation enables reduction to be used as an instantiation of behavioural subtyping.

1 INTRODUCTION

This paper investigates possible definitions of *behavioural subtyping* in the process algebra LOTOS. Interest in this topic is motivated from a number of important areas of current research. Behavioural subtyping impacts on research concerned with,

1. enhancing the specification and development capabilities of process algebra by incorporating features of object oriented methodologies [MC93];
2. providing a theoretical basis for *concurrent OO programming* and models of so called *active objects*, which are objects that exhibit non-uniform service availability [Nie95]; and
3. enhancing existing formal description techniques in order that they can be applied to the new generation of distributed systems, which are typically object oriented [BDLS95].

In all these areas subtyping plays a pivotal role in obtaining incremental system development, with its relationship to different inheritance mechanisms being crucial. The third of these areas has particularly motivated the work presented here. Central to object oriented programming platforms such as CORBA, the TINA DPE (Distributed Processing Environment) and the ODP

*The research presented here has been partially supported by British Telecommunications Research Labs, through their funding of Charles Briscoe-Smith's PhD studentship.

(Open Distributed Processing) Computational Model is the notion of *trading*. A trader is a distinguished object used in order to locate required services. It accepts service offers from objects, and maintains a database of currently available offers. When an object wishes to find a service, it performs an *import* operation on the trader, specifying what kind of service it wants, and receives copies of a number of service offers in reply.

When a client sends a description of the service it wants to the trader, the trader must somehow match this to the offers it has in its database. If it cannot find an offer exactly matching the requested service, it should look for offers of similar services, providing all the facilities that the client wanted, but possibly having other facilities that the client will not use. It is looking for a service which has a superset of the operations the client asked for, which the client could use without knowing that it was any different from the service type it requested. In fact, the relationship between the service requested and the service returned by the trader should be *subtyping*.

The concept of subtyping is familiar from object oriented programming languages[FM94], It is defined as substitutability: type `A` is a subtype of type `B` iff objects of type `A` may be used in any situation where an object of type `B` was expected, without the object's environment being able to tell the difference. Thus, an object of any particular type can *masquerade* as, or stand in for, an object of any of its supertypes. Subtyping is naturally a reflexive and transitive relation, i.e. a preorder.

However, the state of the art in service matching for trading is signature-based subtyping. Unfortunately, such matching is not rich enough to ensure the safety of object interactions in a heterogeneous distributed processing environment. For example, two object types may have methods with the same name but quite different meaning. To take a rather frivolous example, consider the analogy of an artist and a cowboy. Both are able to perform an operation "draw," but the results in each case will be rather different. Thus, it is possible that although signatures match, compatibility in terms of the behaviour of services is not obtained. The insufficiency of purely signature based approaches is witnessed by the increasing interest within OMG for adding behavioural properties to CORBA IDL.

What is actually required is a more powerful interpretation of matching based on (stronger) *behavioural* notions of subtyping (in ODP terms *behavioural compatibility*). Determining suitable interpretations of behavioural subtyping is the subject matter of this paper.

As our notation for describing the behaviour of service types we use the process algebra LOTOS. There are a number of reasons for this choice, not least the role of LOTOS as a formal description technique for open distributed systems and the accepted benefits of the process algebra approach [Mil89]. However, a further benefit of considering LOTOS is that a wealth of correctness relations exist, many of which are related to substitutibility and hence behavioural subtyping. From this domain the testing theories are of particular relevance. In such theories specifications are related if they pass the same tests.

Testing theory is an extremely rich field. In fact it is possible to place the spectrum of process algebra correctness relations (at least those based upon interleaving models of concurrency) in a hierachy of strength, i.e. in terms of their level of discrimination [vG93]. The relative strengths of particular correctness relations is tied to the intrusive capabilities of the tester to observe the specification. In this paper we will use a standard notion of testing in which the tester has the power of a standard LOTOS process (no additional operators are added to the testing language). Since clients in the OO setting will be LOTOS processes this seems a sensible choice. The testing theory induces a preorder that, for the moment, we will call *compatibility*:

S_1 is compatible with S_2 iff for all finite sets of observable actions G and processes P, $S_1 \; |[G]| \; P \stackrel{\sigma}{\Longrightarrow} \approx \mathbf{stop}$ implies $S_2 \; |[G]| \; P \stackrel{\sigma}{\Longrightarrow} \approx \mathbf{stop}$.

where $|[G]|$ is the LOTOS parallel composition operator, $\approx$ is weak bisimulation equivalence, **stop** is the deadlock process, σ is a trace of observable actions and relation composition is denoted by juxtaposition*. This notation will be clarified shortly, but informally, the condition states that S_1 is compatible with S_2 if and only if, for all possible testers, if S_1 can perform a trace σ and then deadlock, then under the control of the same tester S_2 can perform σ and then deadlock. Thus, even more informally, S_1 does not add any new deadlocks to those that can arise from S_2.

In terms of OO and subtyping, in the above definition, G reflects the possible interfaces between S_1 and the tester, i.e. the actions that they can communicate via, and P reflects possible client specifications/programs. We argue that this condition is the basis for an intuitively sensible instantiation of subtyping in the process algebra setting. In OO terms the definition states that,

*S_1 is a subtype of S_2 if and only if any client (tester) using S_1 according to any interface (synchronisation set) can only observe a trace and then observe a deadlock if the client could observe the same trace and a deadlock if it was using S_2 (with the same interface) *.*

From amongst the LOTOS correctness relations, **red** (reduction) is the most important. In particular, modulo handling of divergence, **red** corresponds to failures divergences refinement [Hoa85] and testing preorder [Hen88], which are the principle notions of refinement used in CSP and CCS (respectively). However as it stands, reduction is not a sufficient definition of subtyping. This is because subtyping in the OO context allows *extension of functionality*, e.g. a subtype can offer more operations than its supertype.

*i.e. $S \; |[G]| \; P \stackrel{\sigma}{\Longrightarrow} \approx \mathbf{stop}$ means $\exists Q \, . \, S \; |[G]| \; P \stackrel{\sigma}{\Longrightarrow} Q \; \wedge \; Q \approx \mathbf{stop}$

*Since we test against all possible clients (and not just those that have a subset of the operations of S_2) we get a strong notion of subtyping. We believe that this strength is necessary, e.g. when objects are being concurrently interacted with.

In the process algebra setting extending functionality implies addition of traces. However, reduction enforces a trace subsetting property and thus, does not allow functionality to be extended. In response to this observation a number of previous workers [Rud91] [CRS89] [Nie95] have based their interpretation of subtyping upon an alternative relation: the extension relation (**ext**) [BS86]. However, we will argue against using this relation; rather we will show how to re-interpret LOTOS specifications in order that reduction is the appropriate relation.

Section 2 presents background on LOTOS and outlines how aspects of LOTOS can be related to OO concepts. Section 3 relates the spectrum of LOTOS refinement relations to behavioural subtyping. Section 4 considers the characteristics of behavioural subtyping in OO specification and programming languages and then shows how LOTOS processes can be transformed in order to reflect these characteristics. Then section 5 highlights a simple technique for transforming LOTOS specifications according to this new interpretation. Finally, section 6 summarises and concludes the paper.

2 BACKGROUND

LOTOS. We use a subset of full LOTOS [BB88]:

$$P \ ::= \ \mathbf{stop} \mid a;\, P \mid P\,[]\,P \mid P\,|[G]|\,P \mid \mathbf{choice}\ a \in A\ []\ P \mid X$$

where $a \in \mathbf{Act} \cup \{i\}$ (**Act** contains all observable actions and i is the distinguished hidden action). Thus, our notation has a deadlock process **stop**, action prefix $a;\, P$, binary choice $P\,[]\,P$, parallel composition $P\,|[G]|\,P$, generalised choice **choice** $a \in A\ []\ P$ and reference to a process variable X, through which recursion can be defined. Process definitions have the form, $X \ := \ P$.

We do not include the other basic LOTOS operators, hiding, relabelling, disabling and enabling. This is not because they bring any technical difficulties, but rather to simplify the presentation.

We also assume some semantic constructions. In the following P, P', Q, Q' stand for processes. $\mathcal{L}$ is the alphabet of observable actions associated with a certain process (we will write $\mathcal{L}(P)$ when we need to be explicit about the process we are referring to). The standard semantics for LOTOS [ISO87] map LOTOS processes to Labelled Transition Systems (LTSs) using a structured operational semantics. We will not repeat these inference rules. However, in standard fashion, we denote transitions as: $P \xrightarrow{a} P'$, meaning that P can perform an a and evolve to P'. Furthermore, $\mathcal{L}^*$ denotes traces over $\mathcal{L}$, $\epsilon \in \mathcal{L}^*$ denotes the empty trace and σ ranges over $\mathcal{L}^*$. We assume the following definitions:

$\overset{\epsilon}{\Longrightarrow}$; the reflexive and transitive closure of $\xrightarrow{i}$;
$P \overset{a\sigma}{\Longrightarrow} P'$ iff $\exists Q, Q' \cdot P \overset{\epsilon}{\Longrightarrow} Q \xrightarrow{a} Q' \overset{\sigma}{\Longrightarrow} P'$;

$P \overset{\sigma}{\Longrightarrow}$ iff $\exists P' \cdot P \overset{\sigma}{\Longrightarrow} P'$;

$P \overset{\sigma}{\not\Longrightarrow}$ iff $\neg(\exists P' \cdot P \overset{\sigma}{\Longrightarrow} P')$;

$Tr(P) = \{\sigma \in \mathcal{L}^* \mid P \overset{\sigma}{\Longrightarrow} \}$; the set of traces of P;

$P\ \mathit{after}\ \sigma = \{P' \mid P \overset{\sigma}{\Longrightarrow} P'\}$; the set of states reachable from P by σ;

$\mathit{Ref}(P, \sigma) = \{X \mid \exists P' \in (P\ \mathit{after}\ \sigma) \,.\, \forall a \in X : P' \overset{a}{\not\Longrightarrow} \}$; refusals of $P\ \mathit{after}\ \sigma$.

$\mathit{initials}(P) = \{\ a \in \mathcal{L} \mid a \in Tr(P)\ \}$.

Relating OO Concepts to LOTOS. Before we consider subtyping it is worth clarifying how LOTOS specifications relate to OO concepts. This section highlights some basic relationships.

Class. A class describes the common behaviour of a set of objects. As noted by a number of authors, e.g. [DEBS96] [Smi95] [Rud91], in LOTOS the natural counterpart to a class is a process definition. This describes the common behaviour of instantiations of the process definition.

Object. In OO programming objects are instantiations of a class. Thus, a simple interpretation of instantiation in LOTOS is as process instantiation.

However, more sophisticated interpretations of object instantiation can also be given. For example, [Rud91] [CRS89] interpret instantiation as the LOTOS implementation relation **conf** (which is the LOTOS conformance relation). Thus, any process that conforms to the specification of a class is seen as an instantiation of the class. Although **conf** has a number of undesirable properties as an implementation relation, in principle such an interpretation of instantiation is much richer and more flexible than simple process instantation. In particular, when working in a behavioural setting it seems sensible to interpret instantiation in behavioural terms rather than as a purely syntactic instantiation. Although we will not need to consider this issue of instantiation further in this paper, implicitly instantiations in our setting will be related to their class definition much more strongly than by **conf**; perhaps by testing equivalence.

Operations. The basic units of interaction between objects are operations, also called method invocations, member function calls, or feature calls. In process algebra, the basic units of interaction between processes are actions. The affinity between these two concepts is witnessed by the number of workers in this area who have related the two: [Nie95] [Rud91] [CRS89] [DEBS96] [Smi95].

However, it should be pointed out that this similarity may not be exact, since process algebra actions are considered to be atomic, whereas in many OO models operations have duration. The assumption of atomicity is highly significant in the process algebra setting as it justifies the modelling of concurrency as interleaving. Non-atomic interpretations of actions lead to more complex semantic theories. A simplifying assumption that Nierstrasz makes [Nie95] is only to model method requests. Such an assumption effectively justifies an atomic interpretation of actions when modelling operations. In accordance with this majority of workers we will also enforce a simplifying atomic interpretation of actions/operations.

Finally, the parameters of operations may be modelled using LOTOS's data passing attributes, "!" and "?".

Interface. An object oriented class definition will usually contain a statement of the interface to objects of that class: usually a list of calls which may be made on the objects. The LOTOS equivalent is the set of all non-hidden actions in the process definition.

The above are only the most basic correspondences; there are many more which can be made. For example, Rudkin[Rud91] describes how inheritance and *self* might be introduced into LOTOS and Najm and Stefani [NSF94] consider how object mobility may be obtained. The interested reader is also referred to part IV of [ITU95] which relates OO modelling concepts to LOTOS constructs in the ODP setting.

3 RELATING LOTOS RELATIONS AND SUBTYPING

In this section, we attempt to locate an interpretation of behavioural subtyping from amongst the existing LOTOS correctness relations. Firstly, since subtyping is reflexive and transitive, but not symmetric (a symmetric relation would suggest substitutability in both directions, which is too strong), we will only consider the preorder relations. This choice rules out the equivalences weak bisimulation ($\approx$), strong bisimulation ($\sim$), testing equivalence (**te**) and testing congruence (**tc**) and the implementation relation **conf**, which is not transitive.

Trace Subsetting and Supersetting. We first consider trace preorder, one of the simplest correctness relations. The fact that P_1 is a trace refinement of P_2 is defined as (notice the order that we write refinement, this contrasts with some other workers), $P_1 \leq_{tr} P_2$ iff $\mathrm{Tr}(P_1) \subseteq \mathrm{Tr}(P_2)$. This relation is clearly inappropriate since it does not allow P_1 to have any more traces than P_2, which contradicts the extension of functionality involved in subtyping.

An alternative to $\leq_{tr}$ is trace extension: $P_1 \leq_{tre} P_2$ iff $\mathrm{Tr}(P_1) \supseteq \mathrm{Tr}(P_2)$. This *does* allow new operations to be added and, in fact, is the interpretation of subtyping used in [Pun96]. In Puntigam's work, trace extension serves as a valid check for type safety. Where in this context, type safety ensures that the subtype can understand all operations that the supertype can. However, the relation is not a suitable instantiation of the stronger notion of behavioural subtyping since it allows deadlocks to be added. For example, if X and Y are defined as,

$$X := a;\, \mathbf{stop}\ []\ b;\, \mathbf{stop} \qquad Y := a;\, \mathbf{stop}\ []\ b;\, \mathbf{stop}\ []\ i;\, \mathbf{stop}$$

then $Y \leq_{tre} X$. However, Y is not a behavioural subtype of X. When placed in synchronisation with the process "a; **stop**", X will do action a, but Y may do a, or may do an internal action and then deadlock. If Y deadlocks in a situation where X would not, Y is distinguishable from X and is therefore not a subtype of/compatible with X. In fact, the same criticsm can be levelled at all solely trace based correctness relations, including trace preorder.

Reduction. Reduction[BS86] is a more discriminating refinement relation that adds consideration of liveness properties to trace preorder. Its definition is,

$$P_1 \textbf{ red } P_2 \text{ iff } \mathrm{Tr}(P_1) \subseteq \mathrm{Tr}(P_2\ ) \wedge \forall \sigma \in \mathrm{Act}^* \ .\ \mathrm{Ref}(P_1,\sigma) \subseteq \mathrm{Ref}(P_2,\sigma)$$

(that is, P_1 reduces P_2 iff $P_1 \leq_{tr} P\ _2$ and, after any trace, P_1 does not refuse more than P_2). Interpreting refinement as reduction corresponds to viewing development as reduction of non-determinism. In addition, in terms of our general testing constraint, the property we called compatibility in section 1, we have the following result*:

Theorem 1 *For all processes P_1, P_2, P and $G \subseteq Act$, the following are equivalent:*

1. P_1 **red** P_2
2. $P_1 \ |[G]|\ P \overset{\sigma}{\Longrightarrow} \approx \textbf{stop}$ *implies* $P_2 \ |[G]|\ P \overset{\sigma}{\Longrightarrow} \approx \textbf{stop}$.

Thus, reduction ensures the deadlock property we are seeking. However, as discussed in section 1, it fails to allow extension of functionality. So, as it stands, reduction is not a suitable instantiation of subtyping.

Extension. Since extension[BS86] is sensitive to deadlock properties and supports extension of functionality, it appears at first sight to be an ideal candidate for the subtyping relation. This is witnessed by the large number of workers who have used it as the basis for definitions of subtyping [CRS89] [Rud91] [Nie95]. Its definition is,

$$P_1 \textbf{ ext } P_2 \text{ iff } \mathrm{Tr}(P_1) \supseteq \mathrm{Tr}(P_2) \wedge \forall \sigma \in \mathrm{Tr}(P_2) \ .\ \mathrm{Ref}(P_1,\sigma) \subseteq \mathrm{Ref}(P_2,\sigma)$$

(that is, P_1 extends P_2 iff $P_1 \leq_{tre} P_2$ and, after any trace that P_2 can do, P_1 does not refuse more than P_2). Consider two LOTOS processes, X and Y:

$$X := a;\ \textbf{stop}\ []\ b;\ \textbf{stop} \qquad Y := a;\ \textbf{stop}\ []\ b;\ \textbf{stop}\ []\ c;\ \textbf{stop}$$

Referring to the definition of **ext**, we see that Y **ext** X. Y can do every trace

*This is actually a slightly stronger result than that proved in [BS86], since we do not require trace subsetting between P_1 and P_2. However, this stronger result can be verified with minor changes (involving taking a larger synchronization set) to Brinksma et al's proof.

that X does (and more), and, after any trace that X can do, X refuses at least everything that Y refuses. Conceptually Y defines a class which adds an operation to class X, viz. the action c. Thus, extension enables interface enlargement.

Unfortunately extension does not fulfil our requirements for behavioural subtyping. In particular, extension does not guarantee the definition of compatibility that we gave in section 1. For example, the tester c; **stop** with synchronization set $\{c\}$ serves as a counterexample since,

$$Y \ |[c]| \ c; \mathbf{stop} \overset{c}{\Longrightarrow} \approx \mathbf{stop} \quad \text{but,} \quad X \ |[c]| \ c; \mathbf{stop} \overset{c}{\not\Longrightarrow}$$

Extension only satisfies the following more restrictive theorem, which is proved in [BS86],

Theorem 2 *For all processes P_1, P_2, P; $G \supseteq \mathcal{L}(P_2)$ and $Tr(P_1) \supseteq Tr(P_2)$, the following are equivalent:*

1. P_1 **ext** P_2
2. $\forall \sigma \in Tr(P_2)$, $P_1 \ |[G]| \ P \overset{\sigma}{\Longrightarrow} \approx \mathbf{stop}$ *implies* $P_2 \ |[G]| \ P \overset{\sigma}{\Longrightarrow} \approx \mathbf{stop}$.

Thus, extension only ensures compatibility when restricting to traces of the supertype. However, we require the stronger compatibility property that was highlighted in section 1.

Another way of looking at this problem is that our definition of behavioural subtyping is based on the principle that a subtype must be usable in any situation where the supertype could be used, and not be seen to behave differently. If we have a process which may be a X or a Y, we can detect which it is by trying to perform the action c on the process. If the c is accepted, we have Y, but if c is refused, we must have X. Since it is possible to tell that we have a Y, our definition of behavioural subtyping tells us that Y is not a subtype of X.

Interestingly, this problem with extension is one that Nierstrasz has observed [Nie95]. His illustrative example is that of a one place buffer supertype and a deleting buffer subtype. We can express his example in LOTOS as follows:

$$\mathit{Buf1} := put; get; \mathit{Buf1} \qquad \text{and} \qquad \mathit{DelBuf} := put; (get; \mathit{DelBuf} \; [] \; del; \mathbf{stop})$$

Thus, *DelBuf* behaves as *Buf1* does but it adds the possibility to delete the element in the buffer and then evolve to deadlock. The tester/client which distinguishes the two is analogous to the LOTOS process:

$$T := Prod \, ||| \, Cons \, ||| \, del; \mathbf{stop} \quad \text{with} \quad Prod := put; Prod \quad Cons := get; Cons$$

which yields the composite behaviour shown in figure 1. Now *DelBuf* is clearly

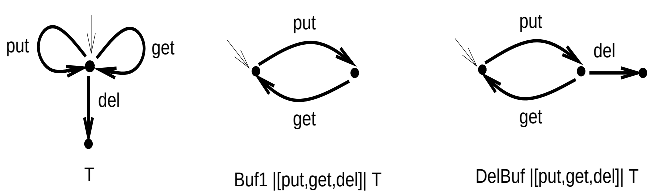

Figure 1 LOTOS Behaviours

an extension of *Buf1*, however, Nierstrasz observes that with the interface $\{put, get, del\}$ and the tester T, *Buf1* cannot reach a deadlock state while *DelBuf* can. Specifically,

$$DelBuf\,|[put, get, del]|\,T \xRightarrow{put\ del} \approx \mathbf{stop} \quad \text{but,} \quad Buf1\,|[put, get, del]|\,T \overset{put\ del}{\not\Longrightarrow}$$

In fact, the problem here is exactly the same as that which we highlighted with behaviours X and Y above. Nierstrasz develops a number of concepts such as *request substitutability* and a notion of *restriction* in order to contain this problem. In contrast, our approach will be to reject extension as an interpretation of behavioural subtyping.

4 FUNCTIONALITY EXTENSION AND UNDEFINED

Undefined Operations in Object Oriented Methods. In order to inform this problem let us consider how functionality extension and particularly adding operations works in OO specification and programming methods.

- *OO Specification Techniques.* A relatively large number of OO specification notions now exist, for example, OO versions of Z, such as Object-Z [Ros92] and ZEST [CR92], OO versions of VDM, such as VDM++ [Lan95] and Liskov and Wing's notation [LW93]. Subtyping is not handled in a uniform way throughout these techniques, so, let us focus on the Liskov and Wing approach which has considered the topic in some depth. In [LW93] a number of conditions are highlighted which must all hold in order to ensure subtyping between a pair of specifications. However, the part of the definition that concerns us here is the pre and post condition relationship between operations. The definition requires that for every operation in the supertype there must exist a corresponding operation in the subtype (although, the subtype may contain extra operations) such that, for corresponding operations, the following holds,

  1. the precondition of the supertype operation implies the precondition of the subtype operation, and

2. the postcondition of the subtype operation implies the postcondition of the supertype operation.

Thus through subtyping, preconditions can be weakened and postconditions can be strengthened. In informal terms, weakening of preconditions enables operations to be applied (i.e. terminate) in more states, while strengthening of postconditions reduces non-determinism. This really does give us what we seek: addition of traces and reduction of refusals when we take subtypes. In spirit, subtyping behaves like refinement in state based specification notations such as Z.
Importantly though, this interpretation of subtyping only works because applying an operation outside its precondition has a very different meaning than the analogous occurrence in process algebra. In process algebras the analogue of applying an operation outside its precondition is the environment trying to perform an action when it is not currently offered, which has the result *deadlock*. In contrast, in state based specification notations such as Z or Liskov and Wing's notation, applying an operation outside its precondition is *undefined*, i.e. is completely unpredictable. In an "operational sense" anything could occur and the choice between these alternatives is non-deterministic.

- *OO Programming Methods.* In strongly typed object oriented systems, it is not possible to call an operation which is not offered by an object. However, other OO systems produce error messages when a program calls an undefined operation, or result in undefined behaviour (such as the program crashing or giving incorrect results), e.g. Smalltalk [GR83].

So, both these OO settings give justification for the argument that attempting to apply an operation that is not currently offered should result in undefined behaviour and not deadlock.

Undefinedness and LOTOS Specifications. What, then, would be the consequence of adapting LOTOS specifications to behave in an undefined fashion if an action that is not currently offered is performed? Unpredictable behaviour can be modelled in LOTOS using non-determinism. In fact, we can highlight the following process:

$$\Omega := (\mathbf{choice}\ a{\in}\mathbf{Act}\ [\,]\ i;\ a;\ \Omega)\ [\,]\ (i;\ \mathbf{stop})$$

which offers a completely non-deterministic behaviour; at every point in its evolution it could offer any action and refuse any set of actions. Since Tr(Ω)=Act* and $\forall\sigma\in$Act*, Ref(Ω, σ)=$\mathcal{P}$(Act), Ω is at the top of the reduction preorder; every behaviour is a reduction of it.

It turns out that we will be able to use reduction as the subtyping relation if the LOTOS definitions of our objects' behaviours are modified using Ω. We will show how the modification is done with two examples.

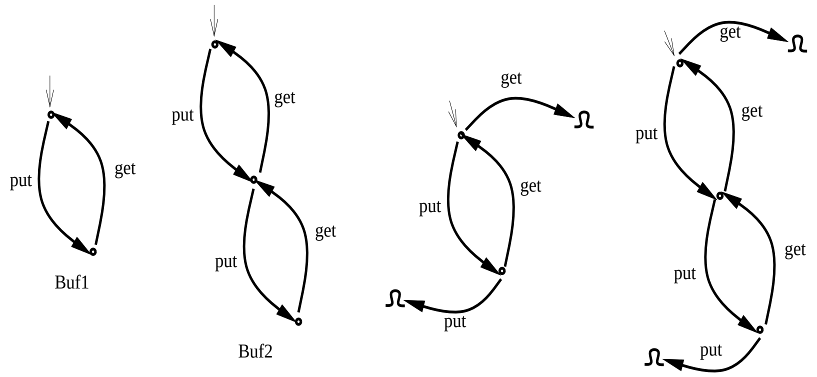

Figure 2 *Buf1* and *Buf2* without and with undefined added

Example 1. A one-place buffer, *Buf1*, was defined earlier. A two-place buffer may be defined:

$$Buf2 := put;\ Buf2a \qquad Buf2a := get;\ Buf2\ []\ put;\ get;\ Buf2a$$

The labelled transition systems corresponding to *Buf1* and *Buf2* are given in figure 2. For these definitions, $\mathcal{L} = \{put, get\}$.

We'd like the two-place buffer to be a subtype of the one-place buffer. Notice that, for the same reasons that we highlighted in our earlier example, as they stand, the two-place buffer is not compatible with the one-place buffer. Thus, to achieve this, we will modify the first two processes as shown in the right hand LTSs of figure 2.

We have added transitions such that every node has at least one transition leading away from it for every possible action in $\mathcal{L}$. Following any of the transitions we have added, the process evolves to Ω (this is in fact a relatively standard technique in process algebra which is used to enable parts of specifications to be extended when refining, see for example [LSW94]).

Using the fact that any behaviour reduces Ω, these two processes are now related in the way we wish; with the addition of undefined behaviour *Buf2* is both a reduction and a subtype of *Buf1*. To justify this, firstly observe that the traces of $\mathcal{T}(Buf2)$ and $\mathcal{T}(Buf1)$ (we will define the mapping $\mathcal{T}$, that adds undefined behaviour shortly) are the same, i.e. $\mathcal{L}^*$. This is because our transformation has ensured that at any state each process "may" perform any action in $\mathcal{L}$. Secondly, observe that for any trace in $\mathcal{L}^*$ the refusals of $\mathcal{T}(Buf2)$ are a subset of those of $\mathcal{T}(Buf1)$. Informally, $\mathcal{T}(Buf1)$ and $\mathcal{T}(Buf2)$ have identical refusals apart from those for traces of the form $put\,put\,\sigma$. For such traces, $\mathcal{T}(Buf1)$ will have evolved to undefined, and will thus refuse everything, while

$\mathcal{T}(\mathit{Buf2})$ may still be performing defined behaviour, in which case it will refuse nothing. Thus, in addition, $\mathcal{T}(\mathit{Buf1})$ is not a reduction/subtype of $\mathcal{T}(\mathit{Buf2})$ since, for example, $\mathcal{T}(\mathit{Buf1})$ can perform the trace *put put* and then refuse anything, while after the same trace $\mathcal{T}(\mathit{Buf2})$ cannot refuse anything.

We introduce some terminology. The original LOTOS specification, i.e. before Ω's have been added, is called *the defined behaviour* of the specification, while the additional choices arising from the addition of Ω's is called the *undefined behaviour* of the specification. We call the LOTOS process resulting from the addition of undefined behaviour the *transformed process*, i.e. $\mathcal{T}$.

Example 2. Interestingly, using the label set $\{put, get, del\}$, when transformed *DelBuf* will be a subtype of *Buf1*. This is because in either of its defined states the transformed *Buf1* can perform a *del* and evolve to Ω. This contrasts with the approach taken in [Nie95], where Nierstrasz attempts to develop conditions that show that in their untransformed form *DelBuf* is not a subtype of *Buf1*.

5 ADDING UNDEFINEDNESS TO LOTOS SPECIFICATIONS

Transforming Specifications. Having introduced the concept of undefined behaviour we have to consider how to add this behaviour to LOTOS specifications in an automated way. There are three possible approaches; we could,

1. leave it in the hands of the specifier to explicitly include the undefined behaviour in their specifications;
2. develop a mapping which takes defined LOTOS specifications and maps them to LOTOS specifications with undefined behaviour; or
3. we could leave the LOTOS specifications unchanged, but rather add the undefined behaviour implicitly at the semantic stage.

Of these three, the first is not a feasible approach as it would make the specifier's task significantly more difficult. The second is feasible, however, defining the mapping is not straightforward. In particular, adding undefined behaviour through the parallel composition operator is quite subtle. Thus, it is the third of these alternatives that we select.

Our approach is to take the LTS of a LOTOS process and derive a new transition relation, which we denote $\vdash\ a\ \rightarrow$. This new transition relation will add states and transitions that reflect the required undefined behaviour. Where $\mathcal{L}$ is the label set of the specification we generate the smallest relation that satisfies the inference rules:

$$R1:\ \frac{P \xrightarrow{a} P'}{P \vdash a \rightarrow P'} \qquad R2:\ \frac{a \in \mathcal{L} \setminus \mathit{initials}(P)}{P \vdash a \rightarrow \Omega}$$

$$R3:\ \frac{a \in \mathbf{Act}}{\Omega \vdash i \rightarrow a;\, \Omega} \qquad R4:\ \frac{-}{a;\, \Omega \vdash a \rightarrow \Omega} \qquad R5:\ \frac{-}{\Omega \vdash i \rightarrow \mathbf{stop}}$$

$R1$ ensures that the new relation contains the relation $\rightarrow$. $R2$ adds the possibility to evolve to Ω when applying an action that is not currently offered. Rules $R3$, $R4$ and $R5$ code up the behaviour of the undefined process Ω.

Non-determinism. These inference rules define a simple means to add undefinedness to an LTS. For deterministic processes, the consequences of applying these rules are very straightforward. For example, the rules will map the first two LTSs of figure 2 to the second two LTSs. However, application of the rules is more subtle in the presence of non-determinism. Consider the following examples of the three archetypal forms of LOTOS non-determinism, with $\mathcal{L} = \{a, b, c\}$:

$$X \ := \ a; b; \mathbf{stop} \ [] \ a; c; \mathbf{stop} \qquad Y \ := \ i; a; \mathbf{stop} \ [] \ i; b; \mathbf{stop}$$
$$Z \ := \ i; a; \mathbf{stop} \ [] \ b; c; \mathbf{stop}$$

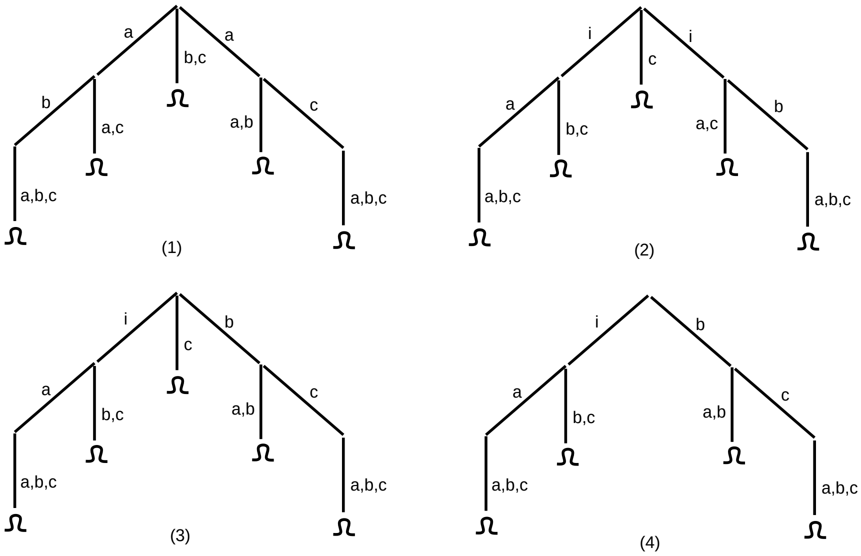

Figure 3 Transformed behaviours

LTSs resulting from adding undefinedness for each of these processes are shown as (1),(2) and (3) of figure 3. This transformation has the virtue of being extremely simple, however, it does not generate the minimum (in terms of least number of transitions) LTS. For example, in (3) the transition labelled c emanating from the start state is in fact redundant and (3) and (4) of figure 3 are testing equivalent.

A consequence of applying this transformation is that (modulo the addition of undefined behaviour) more processes are reductions than they would normally be. For example, once transformed all of the following behaviours would be reductions of Z.

$$a;\ \mathbf{stop}\ []\ b;\ c;\ \mathbf{stop}\ []\ c;\ \mathbf{stop} \qquad a;\ \mathbf{stop}\ []\ c;\ \mathbf{stop} \qquad a;\ \mathbf{stop}\ []\ b;\ a;\ \mathbf{stop}$$

The last of these is perhaps the most suprising as the defined behaviour of the resulting specification requires an a action to be performed after the trace b, while Z requires a c action to be performed after the same trace. However, according to our intuition about subtyping in OO this is correct as the transformed Z can refuse the b action that leads to c, but cannot refuse the b action that leads to Ω. Thus, this situation is only odd if the processes are interpreted without undefinedness.

Further Examples. It is important to note though that while transforming LOTOS specifications in this way yields a more generous relationship between processes, which was after all our original intention, the resulting notion of subtying still remains sensitive to incompatible behaviour.

Consider two examples from [Nie95]: a variable and a non-deterministic stack:

$$\begin{array}{l}
Var := put;\ Var2 \qquad Var2 := put;\ Var2\ []\ get;\ Var2 \\
NDstack := put;\ NDstack2 \\
NDstack2 := put;\ NDstack2\ []\ get;\ NDstack2\ []\ get;\ NDstack
\end{array}$$

Now, it can be checked that, $\mathcal{T}(Var)\ red\ \mathcal{T}(NDstack)$ and $\mathcal{T}(NDstack)\ red\ \mathcal{T}(Buf1)$, but $\neg(\mathcal{T}(NDstack)\ red\ \mathcal{T}(Var))$ and $\neg(\mathcal{T}(Buf1)\ red\ \mathcal{T}(NDstack)$. The latter two of these are because,

$$\begin{array}{l}
Ref(\mathcal{T}(NDstack), put\ get\ get) = \mathcal{P}(\mathcal{L}) \not\subseteq \{\emptyset\} = Ref(\mathcal{T}(Var), put\ get\ get) \\
Ref(\mathcal{T}(Buf1), put\ put) = \mathcal{P}(\mathcal{L}) \not\subseteq \{\emptyset\} = Ref(\mathcal{T}(NDstack), put\ put)
\end{array}$$

In addition, we can handle data passing processes by, in the usual way, expanding full LOTOS into basic LOTOS, using a richer action set and choice to model input alternatives [*]. The following two processes, an infinite stack and an infinite queue, are written in a pseudo full LOTOS.

$$\begin{array}{l}
Stack(l : list) := put?x : nat;\ Stack(x\#l)\ \ []\ \ [l \neq [\,]] \rightarrow get!\,hd(l);\ Stack(tl(l)) \\
Queue(l : list) := put?x : nat;\ Queue(x\#l)\ \ []\ \ [l \neq [\,]] \rightarrow get!lst(l);\ Queue(frnt(l))
\end{array}$$

[*] There are actually some subtleties in how data passing has to be handled which we do not have space to discuss here. One issue is that mapping full LOTOS to basic LOTOS generates a deterministic modelling of output which is not always what is required. Ongoing research is currently seeking to resolve these issues.

where $x\#[x_1,..,x_n] = [x,x_1,..,x_n]$, $\mathit{lst}([x_1,..,x_n]) = x_n$, $\mathit{frnt}([x_1,..,x_n]) = [x_1,..,x_{n-1}]$ and *hd*, *tl* and empty lists, denoted [ ], are treated in the usual way.

As would be expected these two behaviours are incomparable. The following trace/refusal properties demonstrate this (where $\sigma = \mathit{put\_1}\,\mathit{put\_2}\,\mathit{get\_1}$, $\sigma' = \mathit{put\_1}\,\mathit{put\_2}\,\mathit{get\_2}$ and a_v denotes the occurrence of an action at gate a with data value v):

$$\mathit{Ref}(\mathcal{T}(\mathit{Stack}([\,])),\sigma) = \mathcal{P}(\mathcal{L}) \not\subseteq \{\emptyset\} = \mathit{Ref}(\mathcal{T}(\mathit{Queue}([\,])),\sigma)$$
$$\mathit{Ref}(\mathcal{T}(\mathit{Queue}([\,])),\sigma') = \mathcal{P}(\mathcal{L}) \not\subseteq \{\emptyset\} = \mathit{Ref}(\mathcal{T}(\mathit{Stack}([\,])),\sigma')$$

As these examples demonstrate, transformed behaviours have a very precise trace/refusal character. Transformed specifications can perform any trace in $\mathcal{L}^*$ and after all traces either refuse nothing or refuse everything. One consequence of this is that for transformed specifications $\mathit{red} = \mathit{ext} = \mathit{conf}$. This is good news as it has previously been argued [BS86] that checking trace subsetting is a major hindrance to verifying reduction. In fact, this is one of the reasons that Brinksma considered *conf* in the first place. In addition, the normal relationships between the LOTOS equivalences still hold, i.e. $\sim \subset \approx \subset \mathit{te}$.

A Note on Undefined Behaviours. Up to testing equivalence, there are actually several different processes that could be used as Ω. For example, both the following two processes have the same trace/refusal characterisation as Ω.

$$\Omega' := (\textbf{choice}\ a{\in}\textbf{Act}\ [\,]\ i;\ a;\ \Omega')\ [\,]\ (i;\ \textbf{stop})\ [\,]\ (i;\ \Omega')$$
$$\Omega'' := (\textbf{choice}\ a{\in}\textbf{Act}\ [\,]\ a;\ \Omega'')\ [\,]\ (i;\ \textbf{stop})$$

One of the reasons for this is that LOTOS trace/refusal semantics are not sensitive to divergence. Thus although from amongst these processes, Ω' is divergent and Ω and Ω'' are not, the three processes have the same semantic characterisation. The decision not to be sensitive to divergence concurs with the approach taken in bisimulation semantics [Mil89] and is tied to a subtle debate concerning *fair abstraction* [BBK87].

However, it should be pointed out that other models handle this issue differently. For example, in CSP, which employs a chaotic interpretation of divergence, only Ω' would give the most unpredictable behaviour.

6 CONCLUSIONS

A criticism of the approach to adding behaviour that we have presented here is that it is not very refined; a path to undefined is added at any state for any action that is not currently offerred. A more refined approach would allow the specifier to obtain refusal when (s)he wishes and undefined when (s)he wishes. This is an area of ongoing research.

To summarise, then, we have considered the spectrum of LOTOS correctness relations and argued that all fail in some respect to be a suitable instantiation of behavioural subtyping. Then through consideration of how subtyping behaves in OO specification and programming notations we have motivated a re-interpretation of LOTOS specifications in the OO setting. This involves adding undefined behaviour to LOTOS specifications. We have defined a simple LTS based mapping to add undefined behaviour to LOTOS specifications. The main consequence of applying this mapping is that the most well behaved of the LOTOS refinement relations, *reduction, really is behavioural subtyping*.

Acknowledgements. We would like to thank Tim Regan and Steve Rudkin for giving an initial stimulus to this research. In particular, Tim Regan was involved in preliminary discussions from which this paper has evolved. Also thanks to Benjamin Pierce for giving pointers to relevant literature and to Erik Poll and the anonymous referees for useful comments and suggestions.

REFERENCES

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1988.

[BBK87] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of koomen's fair abstraction rule. *Theoretical Computer Science*, 51:129–176, 1987.

[BDLS95] H. Bowman, J. Derrick, P. Linington, and M. Steen. FDTs for ODP. *Computer Standards and Interfaces*, 17:457–479, September 1995.

[BS86] E. Brinksma and G. Scollo. Formal notions of implementation and conformance in LOTOS. Technical Report INF-86-13, Dept of Informatics, Twente University of Technology, 1986.

[CR92] E. Cusack and G. H. B. Rafsanjani. ZEST. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 113–126. Springer-Verlag, 1992.

[CRS89] E. Cusack, S. Rudkin, and C. Smith. An object oriented interpretation of LOTOS. In *Proceedings 2nd International Conference on Formal Description Techniques (FORTE'89)*. North-Holland, December 1989.

[DEBS96] J. Derrick, E.A.Boiten, H. Bowman, and M. Steen. Supporting ODP - translating LOTOS to Z. In *First IFIP International workshop on Formal Methods for Open Object-based Distributed Systems*, Paris, March 1996. Chapman & Hall.

[FM94] Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software (TACS '94), Sendai, Japan*, volume 789 of

LNCS, pages 844–886. Springer, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[ISO87] ISO 8807. *LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, July 1987.

[ITU95] ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.

[Lan95] K. Lano. Specification of distributed systems in VDM++. In *FORTE'95.* Chapman and Hall, 1995.

[LSW94] K.G. Larsen, B. Steffen, and C. Weise. A constraint oriented proof methodology based on modal transition systems. Technical Report RS-94-47, University of Aarhus, 1994.

[LW93] B. Liskov and J. M. Wing. A new definition of the subtype relation. In O. M. Nierstrasz, editor, *ECOOP '93 - Object-Oriented Programming*, LNCS 707, pages 118–141. Springer-Verlag, 1993.

[MC93] A.M.D. Moreira and R.G. Clark. ROOA: Rigorous Object-Oriented Analysis. Technical Report TR 109, Computing Science Department, University of Stirling, Scotland, October 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Nie95] O. Nierstrasz. Regular types for active objects. In *Object-oriented Software Composition*, pages 99–120. prentice-Hall, 1995.

[NSF94] E. Najm, J-B. Stefani, and A. Fevrier. *Introducing Mobility in LOTOS.* ISO/IEC JTC1/SC21/WG1 approved AFNOR contribution, July 1994.

[Pun96] Franz Puntigam. Types for active objects based on trace semantics. In *First IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems*, Paris, March 1996. Chapman & Hall.

[Ros92] G.A. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–78. Springer-Verlag, 1992.

[Rud91] S. Rudkin. Inheritance in LOTOS. In K. R Parker and G. A. Rose, editors, *Formal Description Techniques, IV*, Sydney, Australia, November 1991. North-Holland.

[Smi95] G. Smith. Extending $\mathcal{W}$ for Object-Z. In J. Bowen and M. Hinchey, editors, *9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 276–295. Springer-Verlag, 1995.

[vG93] R.J. van Glabbeek. The linear time - branching time spectrum (I and II). In *Concur'90 and Concur'93, LNCS 458 and LNCS 715.* Springer-Verlag, 1990 and 1993.