ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**D. J. Duke , University of York, UK and A.S. Evans, University of Bradford,
UK (Eds)**

2nd BCS-FACS Northern Formal Methods Workshop

Proceedings of the 2nd BCS-FACS Northern Formal Methods
Workshop, Ilkley, 14-15 July 1997

Translating LOTOS to Object-Z

J. Derrick, E.A. Boiten, H. Bowman and M. Steen

Springer

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Translating LOTOS to Object-Z

John Derrick, Eerke Boiten, Howard Bowman, Maarten Steen

Computing Laboratory, University of Kent

Canterbury, CT2 7NF, U.K.

Phone: +44 1227 764000

Email: J.Derrick@ukc.ac.uk*

Abstract

This paper presents a translation between the formal description technique LOTOS and the object-oriented specification language Object-Z. The need for such a translation lies in the use of formal methods in viewpoint specification, and in particular in the Open Distributed Processing standard. The use of viewpoints as a set of partial interlocking specifications brings an obligation to check the consistency of these partial specifications, and to do so we need to relate specifications written in differing languages. The work presented here supports the consistency checking of viewpoints written using formal methods by defining a translation from full LOTOS to Object-Z. A common semantic model is provided for the languages, and we verify the translation with respect to this model. The translation is illustrated with a small example.

Key words: Object-Z; LOTOS; Open Distributed Processing; Viewpoints; Consistency; Partial Specification.

1 Introduction

In this paper we define and verify a translation between the formal description technique LOTOS [2] and the object-oriented specification language Object-Z [8]. The motivation for deriving such a translation arises from the use of formal methods in viewpoint specification and distributed systems design.

Specification by viewpoints [9] is advocated as a structuring method for the description of complex systems. Each viewpoint represents one perspective of the envisaged system, and thus viewpoints provide a true separation of concerns. In addition, each viewpoint can use a specification language which is dedicated to its particular perspective - acknowledging the generally held belief that no (formal) method applies equally well to all domains of application.

Our motivation for studying viewpoint specification derives from its use in distributed systems design, and in particular in the Open Distributed Processing (ODP) standard [12]. There are five viewpoints, with fixed pre-determined roles, in ODP: *enterprise*, *information*, *computational*, *engineering* and *technology*. Requirements and specifications of an ODP system can be made from any of these viewpoints. For example, the *computational* viewpoint is concerned with the algorithms and data flow of the distributed system function. It represents the system and its environment in terms of objects which interact by transfer of information via interfaces. The *engineering* viewpoint, on the other hand, is more concerned with the distribution mechanisms and the provision of the various transparencies needed to support distribution.

Inherent in any viewpoint approach is the need to check or manage the *consistency* of viewpoints and to show that the different specifications do not impose contradictory requirements [10]. The mechanisms needed to do this depend

*This work was partially funded by the U.K. Engineering and Physical Sciences Research Council under grant number GR/K13035 and by British Telecom Labs., Martlesham, Ipswich, U.K.

on the viewpoint languages used, and we have a particular interest in the use of formal techniques because the ODP reference model places an emphasis on the use of formalism. The reference model includes an architectural semantics which describes the application of formal methods to the specification of ODP systems. Of the available notations, state-based languages such as Z are likely to be used for at least the information, and possibly other, viewpoints. Because ODP has adopted an object-based approach to specifying distributed systems, the object-oriented variant of Z, Object-Z, has been advocated as a language that will meet many of the requirements of ODP viewpoint specification [8, 3]. For the computational and engineering viewpoints, LOTOS is a strong candidate in addition to other, less formal, notations.

Because viewpoints overlap in the parts of the system that they describe, in order to check consistency the relationship between the viewpoints needs to be documented. In simple examples these parts will be linked implicitly by having the same name and type in both viewpoints. In general, however, we may need more complicated descriptions for relating common aspects of the viewpoints, such descriptions are called *correspondences* in ODP [12]. A collection of viewpoints can then be defined to be consistent if and only if a common refinement can be found (i.e. a specification that refines all the original viewpoints) with respect to the correspondences between the viewpoints.

The strategy we envisage to check the consistency of one ODP viewpoint written in Object-Z with another written in LOTOS is as follows. First translate the LOTOS specification to an observationally equivalent one in Object-Z, then use the mechanisms defined in [1, 3] to check the consistency of the two viewpoints now both expressed in Object-Z. These mechanisms attempt to find a common refinement of the two viewpoints - if one exists the viewpoints are consistent.

The aim of the work described here is to support such a consistency checking mechanism by providing a translation of LOTOS into Object-Z. In Section 2 we provide a brief introduction to the languages Object-Z and LOTOS. In order to provide a translation from one language to another it is necessary to be able to verify that the translation is correct, i.e., that the meaning of a term in one language is equivalent to the meaning of that term after translation. The translation we define in this paper is verified against a common semantic model of the two languages, which is introduced in Section 3. This model is based upon the semantics for Object-Z described in [14], which effectively defines a state transition system for each Object-Z specification. This model is also used as a common semantic basis by embedding the standard labelled transition system semantics for LOTOS into it in an obvious manner. Having provided such a basis by which terms in LOTOS and Object-Z can be compared, we are in a position to define the translation between the languages, which we do in Section 4. The verification of the translation algorithm is provided in Section 5. Section 6 illustrates the techniques by providing an example translation.

2 The Languages Object-Z and LOTOS

2.1 Object-Z

Object-Z [8] is an object-oriented extension of the specification language Z [17], which has been developed over a number of years and is the most mature of all the proposals to extend Z in an object-oriented fashion. It has been advocated as one of the languages suitable for use in the ODP viewpoints, particularly in respect of the information viewpoint of the reference model.

Object-Z uses a class schema to encapsulate a state schema together with the operations acting upon that state. It is represented as a named box with zero or more generic parameters. The class schema may include local type or constant definitions, at most one state schema and initial state schema together with zero or more operation schemas. A class may also inherit a number of other classes. The local type and constant definitions of an inherited class are available in the inheriting class. The schemas of an inherited class are either implicitly available or implicitly conjoined with identical named schemas of the inheriting class.

A simple example of an Object-Z class is given by the following:

The variables *count*, *value1* and *value2* declared in the (unnamed) state schema are local to the class. The initial state schema *INIT* defines the initial values of the variables in the state schema. The class specified above has three operations: *a*, *b* and *c*. Each operation has a Δ -list which contains those state variables which may change when the operation is applied to an object of that class. An operation does not change the state variables that are not listed in its Δ -list. Hence the operation *a* implicitly contains a predicate $value2 = value2'$.

The operations in this class allow two integers to be inserted (using *a* and *b*), and the operation *c* will output the maximum of those values. The preconditions of the operations force them to be invoked in a particular order. The interpretation of operations in an Object-Z class differs from that in Z, in that an Object-Z operation cannot occur outside its precondition¹. This interpretation of operation precondition is crucial for the correctness of the translation defined in this paper.

A class can also include instances, i.e. objects, of other classes as state variables. This allows the concise specification of the interaction between components of a system. For example,

¹In Z, an operation is undefined (but enabled) outside its precondition.

<i>Max3</i>
$A, B : M1$
<i>INIT</i>
$A.INIT, B.INIT$
$in1 \triangleq A.a$ $in2 \triangleq A.b$ $in3 \triangleq B.b$ $out \triangleq B.c$ $i \triangleq (A.c[input!/output!] \parallel B.a)$

specifies a class with two state variables, A and B , which are objects of the class $M1$. Initially the objects are in their initial state. The objects have operations applied to them using the dot notation, this notation is made precise in the semantics defined in [14].

The operation i represents an *internal* operation, i.e. one which can be invoked by the object whenever the precondition of i holds, but which cannot be controlled externally². The semantics of internal operations is identical to observable operations, however, weak bisimulation equivalences [13] defined over the semantics will treat internal operations differently to observable operations.

Object-Z provides additional schema operators to those defined in Z. The parallel operator $\parallel$ enables communication between objects to be specified, it behaves like conjunction but also equates inputs and outputs with the same basename [8]. To define the operator, meta-functions $\beta_?$ and $\beta_!$ are used which return the basenames (i.e. apart from the ? and !) of the inputs and outputs respectively. The operation $Op_1 \parallel Op_2$ is then defined as

$$[Op_1][y_1!/y_1?, \dots, y_m!/y_m?] \wedge [Op_2][x_1!/x_1?, \dots, x_n!/x_n?]$$

where $\beta_!(Op_1) \cap \beta_?(Op_2) = \{x_1, \dots, x_n\}$ and $\beta_?(Op_1) \cap \beta_!(Op_2) = \{y_1, \dots, y_m\}$.

In this paper we extend the definition of $\parallel$ with the convention that, in the presence of any type clashes of common variables, $Op_1 \parallel Op_2$ is defined to have predicate *false*.

The notation $Op_1 \bullet Op_2$ denotes enrichment, in that the schema text of Op_1 enriches the environment in which Op_2 is interpreted. That is $Op_1 \bullet Op_2 = [Op_1; d \mid p]$ when $Op_2 = [d \mid p]$ and the free variables of d do not include any variables declared in Op_1 .

2.2 LOTOS

A LOTOS [2] specification of a system defines the temporal relationships among the interactions that constitute the externally observable behaviour of the system. A specification consists of two parts: the *behaviour expression* describes the process behaviour and its interaction with the environment whilst the *abstract data type* (ADT) describes the data structures and value expressions used within the behaviour expression. Basic LOTOS refers to the subset of (full) LOTOS that considers only the temporal aspects of behaviour without value passing or the ADT component.

A simple example of a LOTOS specification is given by the following:

Specification *Max3* [in1, in2, in3, out] : **noexit**
type natural **is**
 sorts *nat*
 opns $0 : \rightarrow nat$

²Not all versions of Object-Z define and use internal operations in the same way, here we use a distinguishing name to denote such operations.

```

    succ : nat → nat
    largest : nat, nat → nat
eqns
    forall x, y : nat
    ofsort nat
        largest(0, x) = x;
        largest(x, y) = largest(y, x);
        largest(succ(x), succ(y)) = succ(largest(x, y));
endtype

behaviour
    hide mid in (Max2[in1, in2, mid] | [mid] | Max2[mid, in3, out])
where
    process Max2[a, b, c] : noexit :=
        a?x : nat; b?y : nat; c!largest(x, y); stop
        []
        b?y : nat; a?x : nat; c!largest(x, y); stop
    endproc
endspec

```

This specification defines a four gate process that accepts three natural numbers at three input gates (in any order), and then offers the largest of them at an output gate. A specification or process behaviour expression is built by applying operators to other behaviour expressions. A behaviour expression may also include instantiations of other processes (e.g. Max2), whose definitions are provided in the where clause of the process definition. The *terminals* of a behaviour expression are the occurrences of the processes *stop*, *exit* or process instantiations (including recursion) within that expression.

The (atomic) observable interactions that a process may engage in are called the *events* or *actions* of that process. An event is thought of as occurring at an interaction point, or *gate*, and in the absence of data passing, the event and gate names coincide. In the above specification the system may interact with its environment via gates *in1*, *in2*, *in3*, *out*. Since *mid* is hidden it does not appear in the gate list. Hidden events give rise to unobservable actions (denoted *i*). Furthermore, the unobservable (or internal) action *i* is also user-definable, in that it can appear directly in a specification, and is used to model the potential non-determinism of a given system. There also exists a special action δ , which is not user-definable, but whose occurrence indicates the successful termination of a process (and the enabling of a subsequent process).

The basic data type specification, as illustrated above, consists of a signature and, possibly, a list of equations. The equations used here define the natural numbers and the function *largest* used to return the maximum of two integer values.

In the translation defined in this paper we will consider the subset of full LOTOS where we have excluded process enabling, disabling and consideration of process exit functionality.

3 The Common Semantic Model

A fully abstract semantics for Object-Z has been presented in [14], along with a number of preliminary models of varying degrees of complexity including the structural and history semantic models. The fully abstract model, a complete-readiness semantics, represents an object as a sequence of events it has undergone, annotated with “ready sets” at each stage in its evolution.

In this paper we will use the structural and history semantic models for Object-Z. The structural model effectively defines a state transition system for each Object-Z class. The history model defines the type of a class in Object-Z in

terms of the set of histories that objects of that class can undergo. The history model, which can be derived from the structural model, is used to provide a meaning for object instantiation and the application of operations to objects. The first two subsections introduce the structural and history models. In Section 3.3 we will use these models to provide a semantics for LOTOS, by embedding the standard labelled transition system semantics of LOTOS into this semantics in an obvious fashion.

3.1 The Structural Model

A full presentation of this model appears in [14]. We will use this work to provide a common semantics for Object-Z and LOTOS, which enables us to verify the proposed translation. In order to embed the standard LOTOS semantics in the structural model, we expand the state space used in the structural semantics, and apart from that the presentation here is identical to that used in [14], which we paraphrase closely.

The structural model views an Object-Z class as consisting of a set of attributes together with a set of operations acting upon those attributes. We assume uniqueness of attributes, operation identifiers and operation parameters.

Let Id be the set of all possible identifiers, $Value$ be the set of all possible values that could be assigned to any identifier of any type, and BE be the set of all LOTOS behaviour expressions. The *state* of the state transition system is either an assignment of values to a set of identifiers or a behaviour expression³, and is defined by ($\mapsto$ denotes a finite partial function):

$$State == (Id \mapsto Value) \cup BE$$

An *event* is defined by the operation's name and a finite partial function giving the values of the operation's parameters:

$$Event == Id \times (Id \mapsto Value)$$

We also use projection functions op and $params$ which are defined by

$$\left| \begin{array}{l} op : Event \rightarrow Id \\ params : Event \rightarrow (Id \mapsto Value) \\ \hline \forall e : Event \bullet e = (op(e), params(e)) \end{array} \right|$$

The structural model of a class, which is represented as a schema $ClassStruct$, defines the semantic model as a state transition system.

$$\begin{array}{|l} \hline ClassStruct \\ \hline states : \mathbb{P} State \\ initial : \mathbb{P} State \\ trans : Event \mapsto (State \leftrightarrow State) \\ \hline \forall s_1, s_2 : states \bullet s_1, s_2 \notin BE \Rightarrow \text{dom } s_1 = \text{dom } s_2 \\ initial \subseteq states \\ \forall e_1, e_2 : \text{dom } trans \bullet op(e_1) = op(e_2) \Rightarrow \\ \quad (\text{dom } params(e_1) = \text{dom } params(e_2) \vee trans(e_i) \subseteq BE \leftrightarrow BE) \\ \forall e : \text{dom } trans \bullet trans(e) \subseteq states \leftrightarrow states \\ \hline \end{array}$$

Within this model the variables represent:

³We could use a disjoint union in the definition of *State* if we wished to stay inside the Z type system, but this clutters the presentation for no extra gain.

states the set of states in the state transition system. This will either be the possible states of an object of an Object-Z class, or set of possible derivatives of a LOTOS behaviour expression.

initial the set of possible initial states of the system. This will either be the set of states which satisfy the predicate of the initial state schema of an Object-Z class, or the initial behaviour of a LOTOS behaviour expression.

trans a function from the set of possible events to the associated set of state transitions. For each event in an Object-Z class, this includes those pairs of states from *states* which satisfy the precondition and resulting postcondition of the operation associated with the event.

For example, in the structural semantics for the Object-Z class *M1* the state space consists of sets assigning values to the three state variables *count*, *value1*, *value2* and thus *states* is given by:

$$states = \{\{count \mapsto n, value1 \mapsto m, value2 \mapsto p\} \mid n, m, p \in \mathbf{N}\}$$

The initial states are those components of *states* which have *count* $\mapsto 0$, i.e.

$$initial = \{\{count \mapsto 0, value1 \mapsto m, value2 \mapsto p\} \mid m, p \in \mathbf{N}\}$$

Finally *trans* contains transitions representing the operations *a*, *b* and *c*, a fragment of which is given by:

$$\begin{aligned} trans = \{ & (a, \{input? \mapsto 10\}) \mapsto \{(\{count \mapsto 0, value1 \mapsto 0, value2 \mapsto 0\}, \\ & \quad \{count \mapsto 1, value1 \mapsto 10, value2 \mapsto 0\}), \dots\}, \\ & \dots, \\ & (c, \{output! \mapsto 21\}) \mapsto \{(\{count \mapsto 2, value1 \mapsto 7, value2 \mapsto 21\}, \\ & \quad \{count \mapsto 2, value1 \mapsto 7, value2 \mapsto 21\}), \dots\} \} \end{aligned}$$

3.2 The History Model

The history model provides a means to define the type of a class in Object-Z, and therefore gives a meaning to object instantiation. The history of an object defines the sequence of states the object has passed through together with the sequence of operations it has undergone, and is defined by the schema *History*.

<i>History</i>	
<i>states</i> : seq _∞ <i>State</i>	
<i>events</i> : seq _∞ <i>Event</i>	
<hr/>	
<i>states</i> ≠ ⟨⟩	
∀ <i>i, j</i> : dom <i>states</i> • <i>states</i> (<i>i</i>) ∉ <i>BE</i> ∧ <i>states</i> (<i>j</i>) ∉ <i>BE</i> ⇒ dom <i>states</i> (<i>i</i>) = dom <i>states</i> (<i>j</i>)	
∀ <i>i</i> : N ₁ • <i>i</i> ∈ dom <i>events</i> iff <i>i</i> + 1 ∈ dom <i>states</i>	

Given a structural model of a class the history model can be derived using a function $\mathcal{H}$ as follows. The first state in the sequence of states of a history of a class is an initial state of the class, and each pair of consecutive states is a possible state transition of the corresponding event in the sequence of events.

$\mathcal{H} : ClassStruct \rightarrow \mathbb{P} History$	
∀ <i>c</i> : <i>ClassStruct</i> •	
$\mathcal{H}(c) = \{h : History \mid h.states(1) \in c.initial \wedge \forall i : \text{dom } h.events \bullet$	
$h.events(i) \in \text{dom } c.trans \wedge (h.states(i), h.states(i+1)) \in c.trans(h.events(i))\}$	

The history model provides a meaning to object instantiation in Object-Z, and to process instantiation and recursion in LOTOS. In Object-Z the meaning of the declaration $p : P$, where *P* is an Object-Z class already defined, is taken

to be $p : \mathcal{H}(\mathbb{P})$ where $\mathbb{P} : ClassStruct$ is the structural model of the class P . We will also use recursive object instantiations (e.g., with the state declaration $A : P$ defined in the class P). The semantics of such recursive definitions is taken to be (as usual) the smallest set of transitions that satisfy the model.

The dot notation in Object-Z for initialising and applying operations to objects can be given a meaning in the history model. The notation $p.INIT$ is identical to the schema which states that the object p has undergone no events. From this it can be deduced that the final state in p 's state sequence satisfies the predicate of the *INIT* schema of its class. For example, from $A.INIT$ in the class *Max3* above, it can be deduced that the final state in A 's state sequence satisfies $count = 0$.

Similarly, if Op is an operation in the class P , then $p.Op$ is represented by a schema which states that the object p undergoes an event associated with the operation Op . It can be deduced (from $\mathcal{H}$) that if $p.Op$ occurs then the state of p before the operation satisfies the precondition of Op , and the state of p after the operation is related to the state before by a state transition defined by Op .

3.3 The Semantics for LOTOS

The standard semantics for LOTOS is an operational semantics given in terms of labelled transitions [2]. For example the labelled transition $P \xrightarrow{x} Q$, where P and Q are behaviour expressions and x is an action, represents the situation where P can perform action x and transform into Q .

Let $Der(P)$ denote the set of derivatives of P , i.e. the set of behaviour expressions that are nodes of the labelled transition tree beginning at P . If G represents the user-definable gates, $Act^+ = G \cup \{i\} \cup \{\delta\}$, and BE is the set of behaviour expressions, then the operational semantics of a basic LOTOS specification defines a labelled transition relation $\longrightarrow \subseteq BE \times Act^+ \times BE$.

To define the structural model of a basic LOTOS specification P we define *states*, *initial* and *trans* from the LTS semantics $\longrightarrow$ of P as follows.

$$initial = \{P\}, \quad states = Der(P)$$

$$\text{dom } trans = \{(a, \emptyset) \mid \exists B_1, B_2 \in Der(P) \bullet B_1 \xrightarrow{a} B_2\}, \quad (B_1, B_2) \in trans(a, \emptyset) \text{ whenever } B_1 \xrightarrow{a} B_2$$

Note that the second component of a transition is the empty-set here because we are at this stage considering basic LOTOS, the second component is used to define the meaning of the value passing within a LOTOS event.

As an example consider $P = a; b; stop$. Then the semantic model is given by

$$initial = \{a; b; stop\}, \quad states = \{a; b; stop, b; stop, stop\}$$

$$trans = \{(a, \emptyset) \mapsto \{(a; b; stop, b; stop)\}, (b, \emptyset) \mapsto \{(b; stop, stop)\}\}$$

The operational semantics of full LOTOS extends that of basic LOTOS by appending a sequence of one or more values to the gate label of the transition. For example, if $P = c!3; stop$, then $P \xrightarrow{c\langle 3 \rangle} stop$, and if $B = a?x : nat; stop$ then for every natural number n there exists a transition $B \xrightarrow{a\langle n \rangle} stop$. The standard semantics of full LOTOS therefore does not distinguish how a value has been generated, i.e., whether it arose because of a variable or value declaration. However, when verifying the translation in the common semantic model we will wish to verify such information. Therefore in the structural model we will represent a transition $B_1 \xrightarrow{a\langle v_1, \dots, v_n \rangle} B_2$ as the structural model transition $(B_1, B_2) \in trans(a, \{1? \mapsto v_1, \dots, n! \mapsto v_n\})$, where the identifiers $?$ and $!$ represent the source of the value. For example, if $B = a?x : nat; stop$ then the structural model will contain the transitions: $(B, stop) \in trans(a, \{1? \mapsto 0\})$, $(B, stop) \in trans(a, \{1? \mapsto 1\})$, $\dots$

3.4 Bisimulation

To prove the translation correct we need to define an appropriate bisimulation equivalence between terms in the semantics. To do so we note that within a particular operation we do not wish to distinguish between parameter names or names of variable declarations, i.e., we consider the external interface of an operation to be given by its name and the values used, not on the names of the parameters. For example, we would like to regard the Object-Z operations $a \hat{=} [x?, y! : \mathbb{N} \mid y! = 2 + x?]$ and $a \hat{=} [u?, v! : \mathbb{N} \mid v! = 2 + u?]$ as equivalent since the external behaviour is identical. The semantics of these operations will give rise to transitions such as $(a, \{x? \mapsto 1, y! \mapsto 3\})$ for the first definition of a and $(a, \{u? \mapsto 1, v! \mapsto 3\})$ for the second definition. Clearly these represent equivalent transitions and are identical apart from the names of the input/output parameters (i.e. the local declarations). Note that in addition the order of declarations in a schema is unimportant, which is why a set such as $\{u? \mapsto 1, v! \mapsto 3\}$ is used in the semantics as opposed to a sequence. Therefore the equivalence we define in this semantic model should not differentiate terms on the grounds of naming or order of the declarations in a transition.

However, in the semantics of LOTOS the order of value and variable declarations in an event is important. For example, $a!7!6$ represents a different event to that described by $a!6!7$. Because the order is important we have used a sequence to represent events in full LOTOS, and an event such as $a!7!6$ will appear as $(a, \{1! \mapsto 7, 2! \mapsto 6\})$ in the semantics.

The purpose of the equivalence we define here is to verify the translation. However, to do so we have to resolve differences in the representation of events: ordered declarations in LOTOS but unordered declarations in Object-Z. Notice that we have already resolved one of the differences between the standard semantics of LOTOS and the semantics of Object-Z, by adding $!$ or $?$ as a subscript to represent the origin of a value as being derived from a value or variable declaration. The standard semantics of full LOTOS would disregard the source of a value and represent $a!7!6$ simply as $a\langle 7, 6 \rangle$. The standard semantics can be recovered if necessary by simply removing the subscripts $!$ and $?$ in the semantics presented here.

The presence of these subscripts allows us to verify that the input (resp. output) to an operation schema has been faithfully translated to a variable (resp. value) declaration in a LOTOS event. The translation will resolve the differences over the order of declarations by adopting an equivalence which disregards the order of declarations, but will require that variable (value) declarations in LOTOS are matched by equivalent input (output) declarations in Object-Z.

To achieve this we will define, for an event e , a set of equivalent events, denoted $[e]$, by setting

$$\begin{aligned} [e] = \{e' \mid & op(e) = op(e') \wedge \# \text{dom } params(e) = \# \text{dom } params(e') \wedge \\ & \forall x, value \bullet x? \mapsto value \in params(e) \text{ iff } \exists y \bullet y? \mapsto value \in params(e') \wedge \\ & \forall x, value \bullet x! \mapsto value \in params(e) \text{ iff } \exists y \bullet y! \mapsto value \in params(e')\} \end{aligned}$$

This definition requires that equivalent events have the same operation name, and that they must also have the same number of parameters, and that each Object-Z input value is matched by some value derived from a LOTOS variable declaration, and finally that each Object-Z output value is matched by some LOTOS value declaration.

We can now define bisimulation. Let B_1 and B_2 be two states in the semantic model. We call B_1 and B_2 strong bisimulation equivalent if there exists a relation $\mathcal{R}$ such that $(B_1, B_2) \in \mathcal{R}$ and $\forall (C_1, C_2) \in \mathcal{R}, \forall e \in Events$

1. $\forall C'_1$ if $(C_1, C'_1) \in trans(e)$ then $\exists f \in [e], C'_2 \bullet (C_2, C'_2) \in trans(f)$ and $(C'_1, C'_2) \in \mathcal{R}$
2. $\forall C'_2$ if $(C_2, C'_2) \in trans(e)$ then $\exists f \in [e], C'_1 \bullet (C_1, C'_1) \in trans(f)$ and $(C'_1, C'_2) \in \mathcal{R}$

Weak bisimulation equivalence is defined in an analogous fashion.

4 The Translation from LOTOS to Object-Z

In this section we define the translation from LOTOS to Object-Z. The ADT component of a LOTOS specification is translated directly into the Z type system (for full details see [4]). To translate the behavioural aspect of a LOTOS specification, we note that there is a strong correlation between classes in object-oriented languages and processes in concurrent systems [18, 11, 14]. We use this correlation as the basis for the translation (which is described in Sections 4.1 and 4.2 below), and map a LOTOS process to an Object-Z class. Adopting this approach allows a natural mapping to be identified between many of the behavioural constructs in the two languages, for example, we find that process instantiation in LOTOS corresponds naturally to object instantiation in Object-Z.

To map a LOTOS process to an Object-Z class we will relate their observable atomic actions, i.e. events in LOTOS and operations in Object-Z. Therefore the translation will map each LOTOS action into an equivalent Object-Z operation schema. For example, a LOTOS specification containing the behaviour $in?x : nat; out!(x + 2); stop$ will be translated into an Object-Z class which contains operation schemas with names in and out . The Object-Z operation schemas have appropriate inputs and outputs to perform the value passing defined in the LOTOS specification. In addition, each operation schema includes a predicate to ensure that it is applicable in accordance with the temporal behaviour of the LOTOS specification.

We begin the translation by defining how specifications are turned into a number of Object-Z classes, each one representing a behaviour expression of the LOTOS specification. The subsequent subsection contains the heart of the translation where the translation of a LOTOS behaviour expression is defined.

4.1 Specifications

A specification S containing type definitions $type_defn_1, type_defn_2$ and process definitions P_1, P_2, P_3 defined by

```

Specification  $S[g_1, \dots, g_n] : \text{noexit}$ 
   $type\_defn_1$ 
behaviour
   $P_1$ 
where
   $type\_defn_2$ 
   $P_2$ 
   $P_3$ 
endspec

```

translates to an Object-Z specification consisting of a collection of type declarations together with a number of Object-Z classes as follows

```

 $T_T(type\_defn_1)$ 
 $T_T(type\_defn_2)$ 
 $T_B(P_3)$ 
 $T_B(P_2)$ 
 $T_B(P_1)$ 

```

Here T_B denotes the LOTOS to Object-Z process definition translation defined by the rules below, and T_T denotes the translation of the LOTOS ADT component into Object-Z type definitions. Each process definition P_i is translated by T_B into a (number of) Object-Z class.

4.2 Process definitions and behaviour expressions

Process definitions involving **where** clauses and type definitions are translated in a similar fashion to specifications. Each process definition is translated into a (number of) Object-Z class, each class corresponding to a behaviour expression.

To translate a process definition we first translate its behaviour expression into an Object-Z class by successively applying the rules given below, working bottom up beginning with the LOTOS terminals, until each operator/terminal has been translated. We then apply the rule for process definitions:

Process definition - Let process $P_B[g_1, \dots, g_n](x_1 : t_1, \dots, x_m : t_m) := B \text{ endproc}$ be a process definition where the behaviour expression B has been translated according to the rules below. Then P_B translates to the Object-Z class

That is we augment the Object-Z class which is the translation of the behaviour expression B into the class P_B , by adding the declarations of fresh variables $x_1 : t_1, \dots, x_m : t_m$ to the state schema of B . $\square$

We can now give the translation rules for behaviour expressions. The rules work with respect to the gate list of the process definition of which the behaviour expression is a part. This is denoted $B[Op_1, \dots, Op_n]$ in the rules that follow. Let P and Q be LOTOS behaviour expressions. We assume that there exist translations of P and Q into Object-Z classes. The variables introduced in a class' state schemas are assumed to be unique with respect to other state variables introduced during the translation of a process. We also assume the existence of a boolean type *bool*.

In the translation rules we have occasion to use a schema $B.STOP$ where B is an Object-Z class. Assume the state schema of B contains the following

Then $B.STOP$ is defined to be the schema

$$[\Delta(s_1, \dots, s_n, A_1, \dots, A_n) \mid \neg s'_1 \wedge \dots \wedge \neg s'_n \wedge A_1.STOP \wedge \dots \wedge A_n.STOP]$$

The purpose of $B.STOP$ is to assert that the class B is disabled, i.e., after $B.STOP$ no operation in B will be enabled. It is necessary to use such an operation in the translation rules for action prefix and choice.

1. Inaction. $B = stop$ translates to the Object-Z class

That is the translation maps a LOTOS process that cannot engage in any action to an Object-Z class with no operations. Both will therefore deadlock. $\square$

2. Termination. $B = exit$ translates to the Object-Z class

The LOTOS behaviour B can perform a δ action and evolve to $stop$ (i.e. it will deadlock). The translation maps this to an Object-Z class which can perform the operation δ , but subsequently can perform no further operations. $\square$

3. Action prefix. $B[a, Op_1, \dots, Op_n] = a ? x : T ! E [pred]; P$ translates to the Object-Z class

where $P.STATE$ denotes the state schema of the class P , and $U = type(E)$.

The temporal ordering defined in the LOTOS behaviour offers action a followed by the ordering defined by P . The translation simulates the same behaviour by using a boolean state variable, t say, and the Object-Z translation of P . Initially, t is true (so the precondition of a holds) but every operation in P is disabled (through $P.STOP$). After a occurs that portion of behaviour is disabled ($\neg t'$), but operations in P are now enabled ($P.INIT$ holds). All operations in the class P are promoted to the class B ($Op_i \triangleq P.Op_i$) to make them available. In addition, P may contain further

occurrences of the operation a , these should be available once the initial a is performed, hence we disjoin $P.a$ to the definition of the operation a .

The LOTOS value and variable declarations are simulated by the input, output and state variables in the Object-Z class. The appearance of T or U in the declarations $Tch_1?$ and $Uch_2!$ are their syntactic representation as a string of characters. This is needed in general for technical reasons, although in practice it is often not necessary.

The rule presented here generalises to an arbitrary number of variable and value declarations in the obvious manner. $\square$

4. Choice. $B[Op_1, \dots, Op_n] = P \parallel Q$ translates to the Object-Z class

B
$P.STATE$ $Q.STATE$
$INIT$ $P.INIT$ $Q.INIT$
$Op_1 \hat{=} (P.Op_1 \wedge Q.STOP) \vee (Q.Op_1 \wedge P.STOP)$ $\vdots$ $Op_n \hat{=} (P.Op_n \wedge Q.STOP) \vee (Q.Op_n \wedge P.STOP)$

The translation of choice makes a copy of both P and Q available in the Object-Z class. Initially, all operations from P and Q are available since both $P.INIT$ and $Q.INIT$ hold. However, once an operation in one branch of the choice is invoked ($P.Op_1$ say), operations from the other branch will be disabled ($\dots \wedge Q.STOP$). This ensures that initially a choice is available between operations from P and Q , but that once that choice is resolved operations from only one class are available. (We have adopted the obvious convention in this paper that if Op is not in the class Q then $Q.Op$ is taken to be false.) This successfully mimics the choice specified in the LOTOS behaviour. $\square$

5. Parallel composition. $B[Op_1, \dots, Op_n] = P \mid [G] \mid Q$ translates to the Object-Z class

B
$P.STATE$ $Q.STATE$
$INIT$ $P.INIT$ $Q.INIT$
$Op_1 \hat{=} (P.Op_1 \vee Q.Op_1)$ $\vdots$ $Op_2 \hat{=} (P.Op_2 \parallel Q.Op_2)$ $\vdots$

where an operation schema definition appears for each operation Op in the gate list of B , and takes the form of that of Op_1 if $Op \notin G \cup \{\delta\}$, and takes the form of that of Op_2 if $Op \in G \cup \{\delta\}$.

The translation of the parallel composition $B[\dots] = P \mid [G] \mid Q$ defines an Object-Z class with operations whose behaviour depends on whether the associated action is in G . If it is not, no synchronisation occurs, and therefore the translation offers a straight choice between, say, $P.Op_1$ and $Q.Op_1$. If it is in G , then the operation can occur precisely

when it occurs in both P and Q . This is achieved by using the Object-Z parallel operator between the two operations, e.g., $P.Op_2 \parallel Q.Op_2$. The full LOTOS value passing synchronisation aspects are also preserved with this operator. $\square$

6. Hiding. $B[Op_1, \dots, Op_m] = \text{hide } g_1, \dots, g_n \text{ in } P$ translates to the Object-Z class

B
$P.STATE$
$INIT$ $P.INIT$
$i \triangleq (P.g_1) \setminus (\text{inouts } g_1) \vee \dots \vee (P.g_n) \setminus (\text{inouts } g_n)$
$Op \triangleq P.Op$

where an operation schema definition of the form $Op \triangleq P.Op$ appears for each operation $Op \in \{Op_1, \dots, Op_m\} \setminus \{g_1, \dots, g_n\}$, and $(\text{inouts } g_i)$ is the list of all input and output parameters of the schema g_i .

Hiding in the context of LOTOS transforms the hidden observable actions of a process into unobservable actions. In the presence of value passing the data is also hidden. In the Object-Z class the hiding of actions is represented by the change of operation name ($i \triangleq (P.g_1) \dots$), and data hiding by hiding both the inputs and outputs ($\dots \setminus (\text{inouts } g_i)$). $\square$

7. Instantiation. Let $B[a_1, \dots, a_n] = P[a_1, \dots, a_n](E_1, \dots, E_m)$, where $P[g_1, \dots, g_n](x_1 : t_1, \dots, x_m : t_m)$ is defined elsewhere. This translates to the Object-Z class (where the identifier A is unique in B)

B
$A : P$
$A.x_1 = E_1 \wedge \dots \wedge A.x_m = E_m$
$INIT$ $A.INIT$
$a_1 \triangleq A.g_1$
$\vdots$
$a_n \triangleq A.g_n$

Process instantiation therefore has a natural counterpart in Object-Z as object instantiation. The identifier used is chosen to be unique because for each process instantiation a new object is instantiated. The substitution of actual gate names for formal gate names is achieved in the translation by operation renaming and promotion ($a_1 \triangleq A.g_1$). The replacement of the parameter list $x_1, \dots, x_m$ by value expressions $E_1, \dots, E_m$ is represented as a predicate equating the variables in the object instantiation to its value ($A.x_1 = E_1 \wedge \dots$). $\square$

The **let** construct used with parametric processes defined by $B[Op_1, \dots, Op_n] = \text{let } x : T := E \text{ in } P(x)$ can be translated as the class

8. Guarding. Let $B[Op_1, \dots, Op_n] = [pred] \longrightarrow P$. This translates to the Object-Z class

The *guard* in the LOTOS behaviour expression means that if *pred* holds the behaviour *P* is possible, otherwise the whole expression is equivalent to *stop*. The Object-Z class generated by *B* exhibits the same behaviour, if *pred* holds the class behaves like *P*, otherwise no operations are enabled, i.e. it deadlocks. $\square$

9. Generalised choice. $B[Op_1, \dots, Op_n] = \text{choice } x : T \llbracket P$ translates to the Object-Z class

The generalised choice offered by the LOTOS behaviour has been mapped here to an unconstrained state variable in the Object-Z class. $\square$

5 Proof of the Translation

In this section we verify that the translation of the behavioural aspects is correct with respect to the common semantic model. The first subsection considers basic LOTOS, which is extended in Section 5.2 to cover full LOTOS. Throughout

this section $\llbracket \cdot \rrbracket$ denotes the semantic map defined earlier, $\llbracket P \rrbracket_L$ denotes the semantics of a LOTOS behaviour expression P , and $\llbracket P \rrbracket_Z$ denotes the semantics of the Object-Z translation of a LOTOS behaviour expression P . The components of these semantics are accessed in the obvious manner, i.e. $\llbracket P \rrbracket_Z.initial$ denotes the *initial* states in the semantics of the Object-Z translation of P . We use $\llbracket P \rrbracket_Z.stop$ to denote the subset of $\llbracket P \rrbracket_Z.states$ that correspond to the states after the application of the schema $P.STOP$. We also define an auxiliary set theoretic operator $\sqcup$ defined on sets of sets, and given by $X \sqcup Y = \{\alpha \cup \beta \mid \alpha \in X, \beta \in Y\}$.

To prove the translation correct we assume that P and Q are LOTOS behaviour expressions with bisimilar Object-Z translations, i.e. $\llbracket P \rrbracket_Z \cong \llbracket P \rrbracket_L$ and $\llbracket Q \rrbracket_Z \cong \llbracket Q \rrbracket_L$. Suppose further that the initial states of P and Q and their translations are in the bisimulation relation, that is, $(\alpha, \beta) \in \mathcal{R}_P$ for all $\alpha \in \llbracket P \rrbracket_L.initial$, $\beta \in \llbracket P \rrbracket_Z.initial$, where $\mathcal{R}_P$ is the bisimulation relation between $\llbracket P \rrbracket_L$ and $\llbracket P \rrbracket_Z$, and similarly for Q .

The proof consists of considering each LOTOS operator in turn, and verifies that the translation rule for that operator is correct. It does this by constructing a relation between a LOTOS behaviour expression B (defined in terms of behaviour expressions P and Q) and its translation into Object-Z in terms of the relations $\mathcal{R}_P$ and $\mathcal{R}_Q$. The relation is shown to be a bisimulation for each LOTOS operator. The reader should refer to [14] for examples of how the semantics $\llbracket B \rrbracket_Z$ is derived for an Object-Z class B .

5.1 Basic LOTOS

In this subsection we prove the translation correct for basic LOTOS. We begin with the LOTOS terminals *stop* and *exit*.

5.1.1 Inaction

Let $B = stop$. The semantics $\llbracket B \rrbracket_L$ is given by $states = initial = \{stop\}$ and $trans = \emptyset$.

The semantics $\llbracket B \rrbracket_Z$ is given by $states = \{\{s \mapsto true\}, \{s \mapsto false\}\}$, $initial = \{\{s \mapsto false\}\}$ and $trans = \emptyset$.

Then the relation $\mathcal{R} = \{(stop, \{s \mapsto false\})\}$ clearly defines a bisimulation between $\llbracket B \rrbracket_L$ and $\llbracket B \rrbracket_Z$.

5.1.2 Termination

Let $B = exit$. The semantics $\llbracket B \rrbracket_L$ is given by $states = \{exit, stop\}$, $initial = \{exit\}$ and $trans = \{(\delta, \emptyset) \mapsto \{(exit, stop)\}\}$.

Whereas the semantics $\llbracket B \rrbracket_Z$ is given by $states = \{\{s \mapsto true\}, \{s \mapsto false\}\}$, $initial = \{\{s \mapsto true\}\}$ and $trans = \{(\delta, \emptyset) \mapsto \{(\{s \mapsto true\}, \{s \mapsto false\})\}\}$.

Then $\mathcal{R} = \{(exit, \{s \mapsto true\}), (stop, \{s \mapsto false\})\}$ clearly defines a bisimulation between $\llbracket B \rrbracket_L$ and $\llbracket B \rrbracket_Z$.

5.1.3 Action prefix

Let $B[a, Op_1, \dots, Op_n] = a; P$.

Then the semantics $\llbracket B \rrbracket_L$ is given by $states = \llbracket P \rrbracket_L.states \cup \{a; P\}$, $initial = \{a; P\}$ and $trans = \llbracket P \rrbracket_L.trans \oplus \{(a, \emptyset) \mapsto \llbracket P \rrbracket_L.trans(a, \emptyset) \cup \{(a; P, P)\}\}$.

The semantics $\llbracket B \rrbracket_Z$ is given by $states = \llbracket P \rrbracket_Z.states \sqcup \{\{t \mapsto true\}, \{t \mapsto false\}\}$, $initial = \{\{t \mapsto true\}\}$ and $trans = \llbracket P \rrbracket_Z.trans \oplus \{(a, \emptyset) \mapsto \llbracket P \rrbracket_Z.trans(a, \emptyset) \cup \{(\alpha, \beta) \mid \alpha \in initial, \beta \in \llbracket P \rrbracket_Z.initial\}\}$

To construct the bisimulation, we add to the bisimulation relation $\mathcal{R}_P$ between $\llbracket P \rrbracket_L$ and $\llbracket P \rrbracket_Z$ the following pairs of states: $(a; P, \alpha)$ where $\alpha \in \llbracket B \rrbracket_Z.initial$. To show that this is still a bisimulation consider the state $a; P$ in $\llbracket B \rrbracket_L$.

From this state the single transition that is enabled is $trans(a, \emptyset) = (a; P, P)$. Now in state α in $\llbracket B \rrbracket_Z$ the only transitions which are enabled are given by $trans(a, \emptyset) = \{(\alpha, \beta) \mid \beta \in \llbracket P \rrbracket_{Z.initial}\}$. By hypothesis we know that $(P, \beta) \in \mathcal{R}_P$ for all such β . This proves one half of the requirements for a bisimulation, the other case is similar.

Hence the relation $\mathcal{R} = \mathcal{R}_P \cup \{(a; P, \alpha) \mid \alpha \in \llbracket B \rrbracket_{Z.initial}\}$ is a bisimulation.

5.1.4 Choice

Let $B[Op_1, \dots, Op_n] = P \sqcap Q$. Then the semantics $\llbracket B \rrbracket_L$ is given by $states = (\llbracket P \rrbracket_{L.states} \setminus \llbracket P \rrbracket_{L.initial}) \cup (\llbracket Q \rrbracket_{L.states} \setminus \llbracket Q \rrbracket_{L.initial}) \cup \{P \sqcap Q\}$, $initial = \{P \sqcap Q\}$ and $trans$ is given by

$$\begin{aligned} trans(a, \emptyset) = & \\ & \llbracket P \rrbracket_{L.initial} \triangleleft \llbracket P \rrbracket_{L.trans}(a, \emptyset) \cup \{(P \sqcap Q, P') \mid P' \in \text{ran}(\llbracket P \rrbracket_{L.initial} \triangleleft \llbracket P \rrbracket_{L.trans}(a, \emptyset))\} \cup \\ & \llbracket Q \rrbracket_{L.initial} \triangleleft \llbracket Q \rrbracket_{L.trans}(a, \emptyset) \cup \{(P \sqcap Q, Q') \mid Q' \in \text{ran}(\llbracket Q \rrbracket_{L.initial} \triangleleft \llbracket Q \rrbracket_{L.trans}(a, \emptyset))\} \end{aligned}$$

The semantics of the Object-Z translation is given by $states = \llbracket P \rrbracket_{Z.states} \sqcup \llbracket Q \rrbracket_{Z.states}$, $initial = \llbracket P \rrbracket_{Z.initial} \sqcup \llbracket Q \rrbracket_{Z.initial}$ and $trans$ is defined by $trans(a, \emptyset) = \llbracket P \rrbracket_{Z.trans'}(a, \emptyset) \cup \llbracket Q \rrbracket_{Z.trans'}(a, \emptyset)$ where

$\llbracket P \rrbracket_{Z.trans'}(a, \emptyset) = \{(X \cup \alpha, Y \cup \beta) \mid \alpha \in \llbracket Q \rrbracket_{Z.states}, \beta \in \llbracket Q \rrbracket_{Z.stop}, (X, Y) \in \llbracket P \rrbracket_{Z.trans}(a, \emptyset)\}$, together with a similar definition for $\llbracket Q \rrbracket_{Z.trans'}$.

To construct the bisimulation, we first relate the initial states, i.e. $P \sqcap Q$ and $\gamma \in \llbracket B \rrbracket_{Z.initial}$. In addition, for each $(P', Y) \in \mathcal{R}_P$, $\beta \in \llbracket Q \rrbracket_{Z.stop}$ we add the pair $(P', Y \cup \beta)$ to the relation so constructed. Similarly, for each $(Q', X) \in \mathcal{R}_Q$, $\alpha \in \llbracket P \rrbracket_{Z.stop}$ we add the pair $(Q', X \cup \alpha)$ to the relation so constructed. The resulting relation is thus given by

$$\begin{aligned} \mathcal{R} = & \{(P \sqcap Q, \gamma) \mid \gamma \in \llbracket B \rrbracket_{Z.initial}\} \cup \\ & \{(P', Y \cup \beta) \mid (P', Y) \in \mathcal{R}_P, \beta \in \llbracket Q \rrbracket_{Z.stop}\} \cup \{(Q', X \cup \alpha) \mid (Q', X) \in \mathcal{R}_Q, \alpha \in \llbracket P \rrbracket_{Z.stop}\} \end{aligned}$$

To show this is a bisimulation consider first the initial states in the two semantics: $P \sqcap Q$ and $\gamma \in \llbracket B \rrbracket_{Z.initial}$. If there exists a transition $(a, \emptyset) \mapsto (P \sqcap Q, P')$ in $\llbracket B \rrbracket_{L.trans}$ then there exists a transition in $\llbracket P \rrbracket_{L.trans}$: $(a, \emptyset) \mapsto (P, P')$, or a transition in $\llbracket Q \rrbracket_{L.trans}$: $(a, \emptyset) \mapsto (Q, P')$. Without loss of generality assume the transition occurs in $\llbracket P \rrbracket_{L.trans}$.

Since $\mathcal{R}_P$ is a bisimulation, there exists Y with $(P', Y) \in \mathcal{R}_P$ and $(P, \gamma) \in \mathcal{R}_P$ for $\gamma \in \llbracket P \rrbracket_{Z.initial}$ such that (γ, Y) is in $\llbracket P \rrbracket_{Z.trans}(a, \emptyset)$. Hence, for all $\alpha \in \llbracket Q \rrbracket_{Z.states}$, $\beta \in \llbracket Q \rrbracket_{Z.stop}$, $(\gamma \cup \alpha, Y \cup \beta)$ is in $\llbracket B \rrbracket_{Z.trans}(a, \emptyset)$ as required.

It remains to show that P' and $Y \cup \beta$ are bisimilar states, but this is clearly the case since $\{(P', Y \cup \beta) \mid (P', Y) \in \mathcal{R}_P, \beta \in \llbracket Q \rrbracket_{Z.stop}\}$ is a bisimulation between the semantics of P as a LOTOS expression and the semantics of its translation into Object-Z.

The remaining cases are proved in a similar fashion.

5.1.5 Parallel composition

Let $B[.] = P \mid [G] \mid Q$. The semantics $\llbracket B \rrbracket_L$ is given by $initial = \{P \mid [G] \mid Q\}$, $states = \{\alpha \mid [G] \mid \beta \mid \alpha \in \llbracket P \rrbracket_{L.states}, \beta \in \llbracket Q \rrbracket_{L.states}\}$ and $trans$ is defined by⁴:

$$\begin{aligned} trans(a, \emptyset) = & \{(U \mid [G] \mid \alpha, V \mid [G] \mid \alpha) \mid (U, V) \in \llbracket P \rrbracket_{L.trans}(a, \emptyset) \wedge \alpha \in \llbracket Q \rrbracket_{L.states}\} \cup \\ & \{(\beta \mid [G] \mid X, \beta \mid [G] \mid Y) \mid (X, Y) \in \llbracket Q \rrbracket_{L.trans}(a, \emptyset) \wedge \beta \in \llbracket P \rrbracket_{L.states}\} \end{aligned}$$

⁴In fact we restrict the transitions to those states that are reachable, but do not write out all the details here as they do not affect the proof.

if $a \notin G$, and if $a \in G$

$$trans(a, \emptyset) = \{(U \mid [G] \mid X, V \mid [G] \mid Y) \mid (U, V) \in \llbracket P \rrbracket_L.trans(a, \emptyset) \wedge (X, Y) \in \llbracket Q \rrbracket_L.trans(a, \emptyset)\}$$

The semantics of the Object-Z translation is given by $states = \llbracket P \rrbracket_{Z.states} \cup \llbracket Q \rrbracket_{Z.states}$, $initial = \llbracket P \rrbracket_{Z.initial} \cup \llbracket Q \rrbracket_{Z.initial}$. The set of transitions is given by the following:

$$\text{If } a \notin G \text{ then } \text{trans}(a, \emptyset) = \{(U \cup \alpha, V \cup \alpha) \mid (U, V) \in \llbracket P \rrbracket_{Z.\text{trans}}(a, \emptyset) \wedge \alpha \in \llbracket Q \rrbracket_{Z.\text{states}}\} \cup \{(\beta \cup X, \beta \cup Y) \mid (X, Y) \in \llbracket Q \rrbracket_{Z.\text{trans}}(a, \emptyset) \wedge \beta \in \llbracket P \rrbracket_{Z.\text{states}}\}$$

If $a \in G$ then $trans(a, \emptyset) = \{(U \cup X, V \cup Y) \mid (U, V) \in \llbracket P \rrbracket_Z.trans(a, \emptyset) \wedge (X, Y) \in \llbracket Q \rrbracket_Z.trans(a, \emptyset)\}$

The bisimulation is defined by $\mathcal{R} = \{(\alpha \mid [G] \mid \beta, X \cup Y) \mid (\alpha, X) \in \mathcal{R}_P, (\beta, Y) \in \mathcal{R}_Q\}$. To show that this is a bisimulation suppose that $(P' \mid [G] \mid Q', X \cup Y) \in \mathcal{R}$ and that $(P' \mid [G] \mid Q', P'' \mid [G] \mid Q') \in \text{trans}(a, \emptyset)$ where $a \notin G$. Then $(P', P'') \in \llbracket P \rrbracket_{L.\text{trans}(a, \emptyset)}$ and $(P', X) \in \mathcal{R}_P$. Since $\mathcal{R}_P$ is a bisimulation, there exists X' with $(P'', X') \in \mathcal{R}_P$ and $(X, X') \in \llbracket P \rrbracket_{Z.\text{trans}(a, \emptyset)}$. Hence $(X \cup Y, X' \cup Y) \in \llbracket B \rrbracket_{Z.\text{trans}(a, \emptyset)}$ and $(P'' \mid [G] \mid Q', X' \cup Y) \in \mathcal{R}$.

The other cases are similarly proved.

5.1.6 Hiding

Let $B[\dots] = \text{hide } g_1, \dots, g_n \text{ in } P$. Wlog we consider the case when $n = 1$. The semantics $\llbracket B \rrbracket_L$ is given by $\text{initial} = \{\text{hide } g_1 \text{ in } P\}$, $\text{states} = \{\text{hide } g_1 \text{ in } \alpha \mid \alpha \in \llbracket P \rrbracket_L.\text{states}\}$. The transitions are given by

For $a \neq g_1$: $(\text{hide } g_1 \text{ in } P', \text{hide } g_1 \text{ in } P'') \in \llbracket B \rrbracket_{L.\text{trans}}(a, \emptyset)$ whenever $(P', P'') \in \llbracket P \rrbracket_{L.\text{trans}}(a, \emptyset)$.

For $a = g_1$: $(\text{hide } g_1 \text{ in } P', \text{hide } g_1 \text{ in } P'') \in \llbracket B \rrbracket_{L.\text{trans}}(i, \emptyset)$ whenever $(P', P'') \in \llbracket P \rrbracket_{L.\text{trans}}(a, \emptyset)$.

The semantics of the Object-Z translation is given by $initial = \llbracket P \rrbracket_Z.initial$, $states = \llbracket P \rrbracket_Z.states$. And if $(\alpha, \beta) \in \llbracket P \rrbracket_Z.trans(a, \emptyset)$ then $(\alpha, \beta) \in \llbracket B \rrbracket_Z.trans(a, \emptyset)$ if $a \neq g_1$ and $(\alpha, \beta) \in \llbracket B \rrbracket_Z.trans(i, \emptyset)$ if $a = g_1$.

The bisimulation is given by $\mathcal{R} = \{(\text{hide } g_1 \text{ in } \alpha, \beta) \mid (\alpha, \beta) \in \mathcal{R}_P\}$. This clearly satisfies the conditions necessary to define a bisimulation.

5.1.7 Instantiation

Let $B[a_1, \dots, a_n] = P[a_1, \dots, a_n]$, where $P[g_1, \dots, g_n]$ is defined previously.

Since $\llbracket B \rrbracket_L$ and $\llbracket P \rrbracket_L$ are bisimilar, and, by hypothesis, so are $\llbracket P \rrbracket_L$ and $\llbracket P \rrbracket_Z$, it remains to show that $\llbracket B \rrbracket_Z$ and $\llbracket P \rrbracket_Z$ are bisimilar. Therefore it suffices to consider the two classes

These two classes are bisimilar (modulo renaming of the operations). To show this a relation between the states of $\llbracket P \rrbracket_Z$ and $\llbracket B \rrbracket_Z$ can be constructed. The states of $\llbracket B \rrbracket_Z$ are given in terms of the possible states of A where $A : \mathcal{H}(\mathcal{P})$, which in turn are provided by the history model. The relation will link a state of $\llbracket P \rrbracket_Z$ to a history of A which has

that state as the final state of its history's state sequence. That is the relation will pair a state $\alpha \in \llbracket P \rrbracket_L.states$ to any history h with $\alpha \in h.states(i)$ where $i = \max(\text{dom } h.states)$.

Then since it can be deduced that an operation $A.g_i$ in B occurs when the state of A before the operation g_i satisfies the precondition of the operation g_i , and the state of A after the operation is related to the state before by a state transition defined by g_i , this relation represents a bisimulation between P and B in this semantics.

5.2 Full LOTOS

In this subsection we sketch how the proceeding proof can be extended to the data passing aspects defined by full LOTOS.

5.2.1 Action prefix

Let $B[a, Op_1, \dots, Op_n] = a?x : T !E [pred]; P(x)$. Then in the semantics $\llbracket B \rrbracket_L$ the action a induces a number of transitions $(B, P(v)) \in \text{trans}(a, \{1? \mapsto v, 2! \mapsto E\})$ for every $v \in T$ such that $pred[v/x]$ is true.

Similarly the semantics $\llbracket B \rrbracket_Z$ will be enriched by a number of transitions $(\alpha, \beta \cup \{x \mapsto v\}) \in \text{trans}(a, \{ch_1? \mapsto v, ch_2! \mapsto E\})$ where $\alpha \in \text{initial}, \beta \in \llbracket P \rrbracket_Z.\text{initial}$ (as defined above) whenever $pred[v/x]$ is true.

Again we can construct a bisimulation between $\llbracket B \rrbracket_L$ and $\llbracket B \rrbracket_Z$ since $(a, \{1? \mapsto v, 2! \mapsto E\}) \in [f]$ if $f = (a, \{ch_1? \mapsto v, ch_2! \mapsto E\})$, and thus the two transitions match up in our generalised definition of bisimulation.

5.2.2 Parallel composition

Let $B[\dots] = P \mid [G] \mid Q$. Consider an action $g \in G$, and suppose that P and Q can synchronise on this action. Then there are three cases to consider: (a) the transition arises as $g!E_1$ in P and $g!E_2$ in Q , (b) the transition arises as $g!E$ in P and $g?x : t$ in Q , or (c) the transition arises as $g?y : t$ in P and $g?x : t$ in Q .

(a) The LOTOS behaviour B performs the event $(g, \{1! \mapsto u\})$ where $u = \text{value}(E_1) = \text{value}(E_2)$. Then the Object-Z translations of P and Q will include the following schema fragments $g \hat{=} [ch_1! \dots \mid ch_1! = E_1 \wedge \dots]$ and $g \hat{=} [ch_1! \dots \mid ch_1! = E_2 \wedge \dots]$ respectively. Therefore the Object-Z translation of B contains $g \hat{=} [ch_1! \dots \mid ch_1! = E_1 \wedge ch_1! = E_2 \wedge \dots]$ by the definition of the operator $\parallel$ in Object-Z. This will induce an equivalent event in the semantics $\llbracket B \rrbracket_Z$ to that in $\llbracket B \rrbracket_L$.

(b) In a similar fashion, B can engage in the event $(g, \{1! \mapsto u\})$ where $u = \text{value}(E)$. Then the Object-Z translations of P and Q will include the following schema fragments $g \hat{=} [ch_1! \dots \mid ch_1! = E \wedge \dots]$ and $g \hat{=} [ch_1? : t \dots \mid x' = ch_1? \wedge \dots]$ respectively. Therefore the Object-Z translation of B contains $g \hat{=} [ch_1! \dots \mid ch_1! = E \wedge ch_1! = x' \wedge \dots]$ by the definition of the operator $\parallel$. This will induce an equivalent event in the semantics $\llbracket B \rrbracket_Z$ to that in $\llbracket B \rrbracket_L$.

(c) In a similar fashion, B can engage in events $(g, \{1? \mapsto u\})$ where $u \in t$. Then the Object-Z translations of P and Q will include the following schema fragments $g \hat{=} [ch_1? : t \dots \mid y' = ch_1? \wedge \dots]$ and $g \hat{=} [ch_1? : t \dots \mid x' = ch_1? \wedge \dots]$ respectively. Therefore the Object-Z translation of B contains $g \hat{=} [ch_1? : t \dots \mid ch_1? = x' \wedge ch_1? = y' \wedge \dots]$ which will induce equivalent events $(g, \{ch_1? \mapsto u\})$ in the semantics $\llbracket B \rrbracket_Z$ to that in $\llbracket B \rrbracket_L$.

Therefore the Object-Z translation preserves the value passing aspects of full LOTOS as well as the temporal ordering.

5.2.3 Hiding

Let $B = \text{hide } g \text{ in } P$. Under hiding, a transition $(P, P') \in \text{trans}(g, X)$ becomes the transition $(B, P') \in \text{trans}(i, \emptyset)$. Similarly in the Object-Z translation of B , a transition $(\alpha, \beta) \in \text{trans}(g, Y)$ will upon hiding the inputs and outputs

of g induce a transition $(\alpha, \beta) \in \text{trans}(i, \emptyset)$, since the hiding of inputs and outputs does not effect the states α, β .

5.2.4 Guarding

Let $B = [pred] \longrightarrow P$. There are two cases to consider: (a) $pred$ holds, or (b) $pred$ doesn't hold. (a) Then $\llbracket B \rrbracket_L.\text{trans} = \llbracket P \rrbracket_L.\text{trans}$, and in the Object-Z translation $Op_i \hat{=} P.Op_i \bullet [pred]$, so that $\llbracket B \rrbracket_Z.\text{trans} = \llbracket P \rrbracket_Z.\text{trans}$. (b) Then $\llbracket B \rrbracket_L.\text{trans} = \text{trans}_{stop} = \emptyset$, and in the Object-Z translation $P.Op_i \bullet [pred] = [false]$, so that $\llbracket B \rrbracket_Z.\text{trans}$ is similarly empty as no operations are enabled.

The other aspects of the full LOTOS translation can be similarly verified.

6 An Example

In this section we illustrate the translation algorithm by translating the LOTOS specification *Max3* given in Section 2.2. This can then be checked for consistency with the Object-Z specification described in Section 2.1. The type definition in the specification *Max3* is translated directly into the Object-Z type system as follows:

$[nat]$

$0 : nat \rightarrow nat$ $succ : nat \rightarrow nat$ $largest : nat \times nat \rightarrow nat$	$\forall x, y : nat \bullet$ $largest(0, x) = x$ $largest(x, y) = largest(y, x)$ $largest(succ(x), succ(y)) = succ(largest(x, y))$
---	---

Any realistic consistency checking toolbox will also contain direct translations from axiomatic descriptions of standard structured types (e.g. sets and sequences) into their Z mathematical toolbox (cf. [17]) equivalents. We will assume that this translation has indeed been made in this example (and hence identify *nat* and $\mathbb{N}$).

To translate the *behaviour* of *Max3*, we first note that it contains two instantiations of the process *Max2*, whose definition is given in the **where** clause of *Max3*. The Object-Z translation will thus contain the definition of the class *Max2* followed by that of *Max3*. Let us consider the class *Max3* first. To translate the behaviour

hide *mid* **in** (*Max2*[*in1*, *in2*, *mid*] | [*mid*] | *Max2*[*mid*, *in3*, *out*])

we begin with the terminals, which here are the process instantiations. The process instantiation rule is applied, producing a class with two instances of the object *Max2*. Subsequently we need to translate the parallel composition (*Max2*[*in1*, *in2*, *mid*] | [*mid*] | *Max2*[*mid*, *in3*, *out*]). Because the only synchronisation is on gate *mid*, all other operations are simply included in the translated Object-Z class, whilst *mid* is defined as $mid \hat{=} (A.c \parallel B.a)$. Finally, *mid* is hidden, and thus we derive the class *Max3* as specified below.

<i>Max3</i>
$A, B : Max2$
<i>INIT</i>
$A.INIT, B.INIT$
$in1 \triangleq A.a$ $in2 \triangleq A.b$ $in3 \triangleq B.b$ $out \triangleq B.c$ $i \triangleq (A.c \parallel B.a)$

To translate *Max2* we follow a similar procedure. Here the terminals are the two instances of *stop*. After translation of the terminals, the rules for action prefix are applied, and finally the rule for choice is used to produce the following class specification (we have simplified the names of the operation parameters for the sake of readability):

<i>Max2</i>
$x, y : \mathbf{N}$ $s_0, s_1, s_2, s_3, t_0, t_1, t_2, t_3 : bool$
<i>INIT</i>
$\Delta(s_0, s_1, s_2, s_3, t_0, t_1, t_2, t_3)$
$s'_0 \wedge \neg s'_1 \wedge \neg s'_2 \wedge \neg s'_3 \wedge t'_0 \wedge \neg t'_1 \wedge \neg t'_2 \wedge \neg t'_3$
$a \triangleq [\Delta(x, s_0, s_1, s_2, s_3, t_0, t_1, t_2, t_3), ch? : \mathbf{N} \mid$ $s_0 \wedge \neg s'_0 \wedge s'_1 \wedge \neg s'_2 \wedge \neg s'_3 \wedge \neg t'_0 \wedge \neg t'_1 \wedge \neg t'_2 \wedge \neg t'_3 \wedge x' = ch?] \vee$ $[\Delta(x, s_0, s_1, s_2, s_3, t_1, t_2, t_3), ch? : \mathbf{N} \mid t_1 \wedge \neg t'_1 \wedge t'_2 \wedge \neg t'_3 \wedge \neg s'_0 \wedge \neg s'_1 \wedge \neg s'_2 \wedge \neg s'_3 \wedge x' = ch?]$ $b \triangleq [\Delta(y, t_0, t_1, t_2, t_3, s_0, s_1, s_2, s_3), ch? : \mathbf{N} \mid$ $t_0 \wedge \neg t'_0 \wedge t'_1 \wedge \neg t'_2 \wedge \neg t'_3 \wedge \neg s'_0 \wedge \neg s'_1 \wedge \neg s'_2 \wedge \neg s'_3 \wedge y' = ch?] \vee$ $[\Delta(y, t_0, t_1, t_2, t_3, s_1, s_2, s_3), ch? : \mathbf{N} \mid s_1 \wedge \neg s'_1 \wedge s'_2 \wedge \neg s'_3 \wedge \neg t'_0 \wedge \neg t'_1 \wedge \neg t'_2 \wedge \neg t'_3 \wedge y' = ch?]$ $c \triangleq [ch! : \mathbf{N}, \Delta(s_2, s_3, t_0, t_1, t_2, t_3) \mid s_2 \wedge \neg s'_2 \wedge \neg s'_3 \wedge \neg t'_0 \wedge \neg t'_1 \wedge \neg t'_2 \wedge \neg t'_3 \wedge ch! = largest(x, y)] \vee$ $[ch! : \mathbf{N}, \Delta(s_0, s_1, s_2, s_3, t_2, t_3) \mid t_2 \wedge \neg t'_2 \wedge \neg t'_3 \wedge \neg s'_0 \wedge \neg s'_1 \wedge \neg s'_2 \wedge \neg s'_3 \wedge ch! = largest(x, y)]$

Although this representation looks complex, it is easily simplified. The complexity has arisen due to the generality needed in the action prefix rule. A simplified version can be given as follows.

<i>Max2</i>
$x, y : \mathbf{N}$ $s : s_0 \mid s_1 \mid s_2 \mid s_3 \mid t_1 \mid t_2 \mid t_3$
<i>INIT</i>
$\Delta(s)$
$s' = s_0$
$a \triangleq [\Delta(x, s), ch? : \mathbf{N} \mid (s = s_0 \wedge s' = s_1 \vee s = t_1 \wedge s' = t_2) \wedge x' = ch?]$ $b \triangleq [\Delta(y, s), ch? : \mathbf{N} \mid (s = s_1 \wedge s' = s_2 \vee s = s_0 \wedge s' = t_1) \wedge y' = ch?]$ $c \triangleq [\Delta(s), ch! : \mathbf{N} \mid (s = s_2 \wedge s' = s_3 \vee s = t_2 \wedge s' = t_3) \wedge ch! = largest(x, y)]$

The translated LOTOS specification can now be compared with the Object-Z specifications of $M1$ and $Max3$ given in Section 2.1, and we can apply the consistency checking techniques as described in [1]. This involves constructing a least refined *unification* of the two viewpoints, in two phases. In the first phase, a unified state schema for the two viewpoints has to be constructed, and this relies on the correspondences between the two viewpoints. The viewpoint operations are then adapted to operate on this unified state. At this stage we have to check that a condition called *state consistency* is satisfied. In the second phase, called *operation unification*, pairs of adapted operations from the viewpoints which are linked by a correspondence have to be combined into single operations on the unified state. This also involves a consistency condition (*operation consistency*) which ensures that the unified operation is a refinement of the viewpoint operations.

It is easily seen that the LOTOS and Object-Z specifications are consistent if $M1$ is consistent with $Max2$, that is they are consistent if we can find a common refinement of $M1$ and $Max2$. After making the obvious correspondences it can be shown that it is possible to find a common refinement of the two classes and that the two classes (and therefore the two specifications) are indeed consistent.

7 Conclusions

Using viewpoints written in process algebras and state-based languages requires that the gap between different specification styles is bridged. To do so we have used an object-oriented variant of Z which has a natural behavioural interpretation. It is this behavioural interpretation which makes it possible to define a state transition system for Object-Z specifications. We used this state transition system as a common semantic model for the two languages, and thereby defined and verified a translation between LOTOS and Object-Z.

Related work includes [15, 11] where methods of formally specifying concurrent systems using Object-Z together with CSP are developed. However, the motivation there is not consistency checking between viewpoints, but rather the construction of one specification using a combination of two languages. The basis of the language integration defined in [15, 11] is a semantics of Object-Z classes identical to that of CSP processes, where classes are related to processes and events to operations in a similar manner to the work described here. The treatment of input and output parameters of operations is, however, slightly different leading to a different treatment of refinement [16]. The relationship between the Z and LOTOS refinement relations in the context of consistency checking in ODP is discussed in [7, 6], where the latter develops refinement relations for Z specifications that contain internal operations.

The work described in this paper builds upon earlier work described in [5] which provided a partial translation between LOTOS and Z. However, this was defined via a complex intermediate semantic model, and without a full treatment of instantiation and recursion. The direct translation defined here has the benefit of preserving some of the syntactic structure of a LOTOS specification upon translation. For example, process instantiation can be translated directly to Object-Z object instantiation. The translation also sheds light on how behavioural specification is structured in the two languages. Consider, for example, parallel composition. In LOTOS a parallel composition is formed between two complete behaviours, as in $P \parallel Q$. However, in Object-Z we can't form such a parallel composition between *classes*, rather we compose *operations* together using the Object-Z parallel composition schema calculus operator, as in $P.Op \parallel Q.Op$. Thus the translation of $P \parallel Q$ has to be given as one explicit class definition, but the behaviour inherent in $P \parallel Q$ appears in the operation definitions as $Op \hat{=} P.Op \parallel Q.Op$. Thus the translation preserves structure, but in Object-Z that structure appears at an operation level rather than a class or behavioural level.

If P is not a process instantiation simpler translation rules can be given, for example, in translating a behaviour such as $a?x : nat; b?y : nat; c!largest(x, y); stop$ (part of the definition of $Max2$) intermediate variables can be used to translate the action prefixes directly to the simpler version of $Max2$ given in Section 6. Similar simplifications can be given for the LOTOS choice operator when, in translating $P \sqcap Q$, P and Q are not process instantiations. Further work to be done in this area includes development of less complex translation rules for these situations, particularly for behaviours involving action prefix and choice. More information about the work described here can be found at: <http://alethea.ukc.ac.uk/Dept/Computing/Research/NDS/consistency>.

8 References

- [1] Boiten E, Derrick J, Bowman H, Steen M. Consistency and refinement for partial specification in Z. In: Gaudel M-C, Woodcock J (eds) FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Methods Europe. Springer-Verlag, 1996, pp 287-306 (Lecture Notes in Computer Science No. 1051)
- [2] Bolognesi T, and Brinksma E. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.
- [3] Bowman H, Derrick J, Linington P, Steen M. FDTs for ODP. *Computer Standards and Interfaces*, 17:457–479, September 1995.
- [4] Derrick J, Boiten E, Bowman H, Steen M. Cross viewpoint consistency in Open Distributed Processing, Part 2 - Translating LOTOS to Z. Technical report, Computing Laboratory, University of Kent at Canterbury, 1996.
- [5] Derrick J, Boiten E, Bowman H, Steen M. Supporting ODP - translating LOTOS to Z. In E. Najm and J.-B. Stefani, editors, *First IFIP International workshop on Formal Methods for Open Object-based Distributed Systems*, pages 399–406, Paris, March 1996. Chapman & Hall.
- [6] Derrick J, Boiten E, Bowman H, Steen M. Weak refinement in Z. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z formal specification notation*, LNCS 1212, pages 369–388, Reading, April 1997. Springer-Verlag.
- [7] Derrick J, Bowman H, Boiten E, Steen M. Comparing LOTOS and Z refinement relations. In *FORTE/PSTV'96*, pages 501–516, Kaiserslautern, Germany, October 1996. Chapman & Hall.
- [8] Duke R, Rose G, and Smith G. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, September 1995.
- [9] Finkelstein A, Spanoudakis G (eds). SIGSOFT '96 International Workshop on Multiple Perspectives in Software Development (Viewpoints '96). ACM, 1996
- [10] Finkelstein A.C.W., Gabbay D, Hunter A, Kramer J, and Nuseibeh B. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.
- [11] Fischer C. CSP-OZ - a combination of CSP and Object-Z. In H. Bowman and J. Derrick, editors, *Second IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 423–438. Chapman & Hall, July 1997.
- [12] ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.
- [13] Milner R. *Communication and Concurrency*. Prentice-Hall, 1989.
- [14] Smith G. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
- [15] Smith G. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In *Formal Methods Europe (FME '97)*, 1997. To appear.
- [16] Smith G and Derrick J. Refinement and verification of concurrent systems specified in Object-Z and CSP. In *First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, 1997. To appear.
- [17] Spivey J.M. *The Z notation: A reference manual*. Prentice Hall, 1989.
- [18] Yonezawa A and Tokoro M. *Object-Oriented Concurrent Programming*. MIT Press, 1987.