Viewpoint Consistency in ODP

Eerke Boiten, Howard Bowman, John Derrick, Peter Linington, Maarten Steen
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Email: E.A.Boiten@ukc.ac.uk.)

February 19, 2011

Abstract

Open Distributed Processing (ODP) is a joint ITU/ISO standardisation framework for constructing distributed systems in a multi-vendor environment. Central to the ODP approach is the use of viewpoints for specification and design. Inherent in any viewpoint approach is the need to check and manage the *consistency* of viewpoints.

In previous work we have described techniques for consistency checking, refinement, and translation between viewpoint specifications, in particular for LOTOS and Z/Object-Z. Here we present an overview of our work, motivated by a case study combining these techniques in order to show consistency between viewpoints specified in LOTOS and Object-Z.

Keywords: Open Distributed Processing; Viewpoint Consistency; Formal Methods; Object-Z; LOTOS.

1 Introduction

There have been dramatic recent developments in the area of distributed computing, e.g. applications involving multi-media made possible by high speed networks; however, for the next generation of global distributed systems to be realised, significant system engineering problems must be resolved. Examples of such problems include the wealth of legacy components that must be accommodated, software interoperability, the heterogeneity of underlying technologies and the sheer complexity of such *distributed* systems.

The *Open Distributed Processing* (ODP) architecture is a vehicle for addressing these system engineering problems. Open Distributed Processing (ODP) is a joint ITU/ISO standardization framework for constructing distributed systems in a multi-vendor environment. The architecture has reached a level of maturity, with the main ODP document, the Reference Model for Open Distributed Processing (RM-ODP), recently progressed to become an international standard. The interested reader is referred to both published introductions [**?**, **?**, **?**] and the standards documents themselves [**?**].

Significant features of ODP include *object based* specification and programming, use of *transparencies* to hide aspects of distribution and *viewpoints* [**?**]. The latter of these is of particular significance here; it provides a basic separation of concerns, enabling different participants to observe the system from suitable perspectives and at suitable levels of abstraction. It is a central device for structuring and managing the complexity inherent in describing systems. Its value in this respect is witnessed by the work of the Telecommunications Information Networking Architecture (TINA) group [**?**], which is using ODP viewpoints extensively in its work on defining a software architecture for the next generation of telecoms systems.

The use of viewpoints in ODP mirrors their increasing importance throughout software engineering, e.g. in requirements engineering [**?**], OO design methodologies [**?**], formal system development [**?**] and in software engineering in general [**?**]. The idea is that rather than having a single thread of system development, in the style of the classic waterfall approach, multiple partial specifications (i.e. viewpoints) of a system are considered. Each particular specification represents a different perspective on the system under development and, in fact, may well be written by a different specifier. ODP uses five predefined viewpoints - *the enterprise viewpoint*, *the information viewpoint*, *the computational viewpoint*, *the engineering viewpoint* and *the technology viewpoint*. (Predefinition of viewpoints in ODP contrasts with how they arise in other models, e.g. [**?**] where specifiers can introduce new viewpoints as they wish.) ODP is not prescriptive about the choice of specification language to be adopted with particular viewpoints. However, it does advocate that the chosen languages should be formal [**?**].

One of the consequences of adopting a multiple viewpoints approach to development is that descriptions of the same or related entities can appear in different viewpoints and must co-exist. Thus, different viewpoints can impose contradictory requirements on the system under development and *consistency* of specifications across viewpoints becomes a central issue. The problem is complicated by the fact that we can expect viewpoint specifications to be written in different languages (viz. languages particularly suited for the viewpoint at hand, e.g. Z [**?**] for the information viewpoint and LOTOS [**?**, **?**] for the engineering viewpoint [**?**]).

Thus, providing techniques to check viewpoint consistency is one of the major research topics surrounding ODP viewpoints modelling and a number of workers have responded to this challenge, [**?**, **?**, **?**]. In particular, a research project, *Cross Viewpoint Consistency in Open Distributed Processing*[1] was undertaken at the University of Kent at Canterbury in order to explore this issue. The scope of this project was broad, including theoretical investigations of the nature of consistency checking [**?**, **?**], techniques for consistency checking within specific formalisms, e.g. LOTOS [**?**, **?**, **?**] and Z [**?**, **?**] and, techniques for consistency checking across specification languages [**?**, **?**].

This paper reviews the results of this project through the presentation of a worked consistency checking example. The example is the specification of CCITT's *Signalling System No. 7 Protocol* [**?**] from two viewpoints: engineering and computational. We give a computational viewpoint specification in Object-Z and an engineering viewpoint description containing LOTOS and Object-Z fragments.

The paper illustrates the scope of the results of our project by presenting a complete consistency check between these viewpoints. First we show how the Object-Z and LOTOS fragments can be reconciled by translating the LOTOS fragments into observationally equivalent Object-Z ones. Then we check the consistency of the two viewpoint specifications now both expressed in Object-Z. The constructive method used for this results in a common refinement of the two Object-Z specifications, whose existence demonstrates consistency of the original viewpoints.

It is important to note that although our example is couched in terms of the computational and engineering viewpoints the techniques we present are general in nature. They are techniques for checking the consistency between any set of viewpoint specifications, in fact, any set of partial specifications written in LOTOS and/or (Object-)Z. For example, in [**?**] they are applied to the ODP information and enterprise viewpoints.

The paper is structured as follows. First we present some background on the ODP model and consistency in section **??**. Section **??** introduces the LOTOS and Object-Z specifications of the computational and engineering viewpoints. Section **??** describes how we relate specifications in LOTOS with Object-Z specifications. Section **??** describes the basic principles of consistency checking. Consistency checking in Z and Object-Z are reviewed in section **??** and consistency

[1]Funded by the UK Engineering and Physical Sciences Research Council under grant GR/K13035 with additional support from British Telecom Research Labs, Martlesham, Ipswich, U.K..

checking in LOTOS is considered in section **??**. Both these sections use the running example to illustrate the techniques introduced. Section **??** discusses issues for further work and the potential for tool support and finally, section **??** discusses related work.

2 ODP Reference Model

2.1 Overview and Motivation

The initiative which lead to the standardization of Open Distributed Processing [**?**] came from a growing awareness that many of the communications-oriented standardization activities aimed at the provision of Open Systems Interconnection required a broader framework than was provided by the OSI Reference Model [**?**]. Standardization of distributed applications such as interpersonal messaging or transaction processing requires a view of the way many components are linked into a distributed system, and of the resources and structures used by these components. A simple interconnection model is not powerful enough. What is needed is a model which can combine the description of system structure with statement of system-wide objectives and constraints, so that the adequacy of the solutions proposed can be judged against the system's original purpose. The ODP standardization initiative is a response to these issues.

The central ODP document is the Reference Model for Open Distributed Processing (RM-ODP) [**?**]. This aims to provide a unifying framework for the standardization of any mechanized distributed system, and of the supporting models, techniques and notations needed to describe such a system. Its scope is very broad, including support for all types of traditional data processing systems, networked personal computers, real-time systems and multimedia systems.

As indicated earlier, the reference model recently progressed to become an international standard. Consequently, the focus of research within the ODP community has now moved from defining the reference model to standardizing components of the architecture, e.g. the trader [**?**] and the enterprise language [**?**] and in addition, to progressing specific instantiations of the architecture. Examples of such instantiation include OMG's CORBA [**?**] and Microsoft's D-COM [**?**] architectures; both of which can be viewed as instantiating the ODP Computational and Engineering viewpoints.

2.2 ODP Viewpoints

The complete specification of any non-trivial distributed system involves a very large amount of information. Attempting to capture all aspects of the design in a single description is generally unworkable. Most design methodologies aim to establish a coordinated, interlocking set of models each aimed at capturing one facet of the design, satisfying the requirements which are the concern of some particular group involved in the design process. In ODP, this separation of concerns is established by identification of five *viewpoints* [**?**].

For each of the five viewpoints, the reference model presents a set of definitions and a set of rules which constrain the ways in which the definitions can be related.

- The **enterprise viewpoint** is concerned with business policies, management policies and human user roles with respect to the systems and the environment with which they interact. Use of the word enterprise here does not imply a limitation to a single organization. The model constructed may well describe the constraints placed on the interaction of a number of distinct organizations.

- The **information viewpoint** is concerned with information modelling. By factoring an information model out of the individual components, it provides a common view which can be referenced by the specifications of information sources and sinks and the information flows between them. The information viewpoint defines concepts for information schema definition. The viewpoint distinguishes between an instantaneous view of information (a static schema), a statement of information which is necessarily unchanged by the system (an invariant schema) and a description of information reflecting the behaviour and evolution of the system (a dynamic schema).

- The **computational viewpoint** is concerned with the algorithms and data flows which provide the distributed system function. This viewpoint specifies the individual components which are the sources and sinks of information flows. It represents the system and its environment in terms of objects which interact by transfer of information via interfaces. This does not necessarily imply that the computational objects will be realized in the eventual system by separate components, but indicates which are the candidate objects from which components can be chosen.

- The **engineering viewpoint** is concerned with the distribution mechanisms and the provision of the various transparencies needed to support distribution. The engineering viewpoint defines a number of functional building blocks which can be combined together to provide the requested transparencies (e.g. distribution, failure or migration transparencies). The engineering viewpoint lists a large number of supporting functions which are candidates for standardization (or for which there are already standards) and gives initial definitions of them.

- The **technology viewpoint** is concerned with the detail of the components and links from which the distributed system is constructed.

It is important to note that ODP is not specific about the choice of specification language that should be applied in particular viewpoints, rather the reference model gives an abstract definition of the set of basic constructs that would be used when describing systems from that particular viewpoint. Then the reference model advocates a process of instantiation by which existing specification notations are related to these constructs; the ODP architectural semantics takes a first step in this direction [**?**].

Furthermore, since different languages are applicable to the specification requirements of different viewpoints, e.g. LOTOS for the engineering viewpoint and Z/Object-Z for the information viewpoint, it is clear that we must work within a multiple language setting. In particular, consistency checking must be performed both within and across specific languages.

2.3 Formal Description in ODP

Formal description has been extensively employed in Open Distributed Processing, [**?**, **?**, **?**, **?**, **?**, **?**]. Within ODP, formal description is viewed as enabling precise, unambiguous, and abstract definition and interpretation of ODP standards. This is the familiar motivation for employing FDTs in standardization activities. However, the spectrum of FDT usage in ODP is both extensive and diverse. Which FDT should be employed for each particular role is a central issue. The spectrum of available FDTs also offers significant diversity. For example, LOTOS [**?**, **?**], Estelle [**?**] and SDL [**?**] are targeted at issues of explicit concurrency and interaction (specifying ordering and synchronisation of abstract events). Communication protocols are a typical example of this class of application. In contrast, approaches such as Z [**?**] and VDM [**?**] address specification of software systems in terms of data state change. Importantly, none of these FDTs fully address the specification requirements of modern distributed systems and Open Distributed Processing

in particular. Such systems are extremely broad, encompassing, for example, both information modelling and description of engineering infrastructures.

The paper [**?**] makes a broad assessment of the role of different FDTs within ODP. In the current paper however, we will focus on two basic techniques: LOTOS and Z (in fact, an object oriented variant of Z, called Object-Z). These two FDTs are chosen because they are conceptually two of the most different specification notations being considered in ODP. Thus, if we can successfully reconcile these two specification notations we will have a strong indicator of the effectiveness of our techniques.

2.4 Consistency and Correspondences in ODP

In order to be able to check the consistency of multiple viewpoint specifications we first need to define what is meant by consistency - at one time the ODP reference model alluded to three different definitions. However, this can be resolved by adopting a formal framework as described in [**?**]. This provides a definition of consistency between viewpoints general enough to encompass all three ODP definitions.

Correspondences. Because viewpoints overlap in the parts of the envisaged system that they describe, we need to describe the relationship between the viewpoints. 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. We will introduce a number of examples of correspondences in section **??**.

Some correspondences follow directly from the ODP reference model. These are mostly consistency requirements of the structural type, similar to those occurring in diagrammatic methods for requirements engineering [**?**]. For example, the model requires that each computational object which is not a binding object correspond to a set of one or more basic engineering objects (with connecting channels). However, if full consistency (i.e. conformant behaviour) is required, such correspondences need to be more detailed, stating which behaviours are viewed as corresponding between the viewpoints.

The traditional way to check for consistency is by translating all the viewpoints to a common underlying semantic framework (e.g. first order predicate logic [**?**]) – if all of these translations have a common model (their conjunction is satisfiable) then the viewpoints were consistent. However, this approach suffers from traceability problems, as discussed in more detail in [**?**]: a reported inconsistency will be in terms of the semantic framework, and can normally only be partially translated back into the original languages. In any case, it will be hard to recover all of the original syntactic structure in such a back-translation. Additionally, the use of an underlying common model, also due to loss of syntax, makes incremental consistency checks (on a specification in development) more difficult.

Development Relations. An alternative approach to consistency checking has been described in our papers [**?**, **?**, **?**]. Consistency of viewpoint specifications is defined as the existence of a common "implementation". Checking this directly would be just another "common model" approach. However, ideally, abstract formal specifications are developed in small formal steps, characterised by *refinement* relations. In order to gather refinement, implementation, and translation (to another formal notation) under one name, we use the term "*development relation*". For demonstrating the existence of a common *implementation*, finding a common *development* is sufficient. If both specifications use the same development relation, and the development relation is complete (in the sense that all implementations can be obtained by development), existence of a common

Figure 1: Consistency as the existence of a common implementation

development is even *necessary*. Furthermore, under certain conditions common developments of viewpoint specifications can be used to check consistency with additional viewpoints, ensuring that all consistency checks can be just between two specifications.

Thus, our definition of consistency will be:

A set of viewpoint specifications are consistent if there exists a specification that is a development of each of the viewpoint specifications with respect to the identified development relations and the correspondences between viewpoints. This common development is called a unification.

As just indicated, different viewpoint specifications may be related to the unification by different development relations. For example, the LOTOS engineering viewpoint might be related by a conformance relation, while the Z computational viewpoint might be related by the Z refinement relation. (Note also that the definition makes no reference to the five ODP viewpoints – in particular, it allows viewpoint decomposition to be applied recursively, *within* the ODP viewpoints, as well.)

Least Developed Unification. Besides a definition of consistency, we have also investigated methods for constructively establishing consistency [**?**]. This involves defining algorithms which build unifications from pairs of viewpoint specifications. An important notion in this context is that of a *least developed unification*. This is a unification such that all other unifications are developments of it. Thus, it is the *least developed* of the set of possible unifications according to the development relations of the different viewpoints.

Using least developed unifications as intermediate stages, global consistency of a set of viewpoints can be established by a series of binary consistency checks.

Unfortunately, it is not the case that least developed unifications can always be derived. In [**?**] properties that development relations must possess for such unifications to exist are investigated. In most cases development relations possess the required properties (in particular, for Z refinement, least developed unifications can always be constructed) and as a reflection of this we will use a strategy of constructing successive least developed unifications below in order to check the consistency of the protocol viewpoints.

Techniques Considered. This definition of consistency is general in nature and can be instantiated in many ways within and between languages. In this paper we present an example which demonstrates how consistency can be shown between one viewpoint specified in LOTOS, and two others specified in (Object-)Z.

Figure 2: Global consistency from binary consistency

3 A Worked Example

We illustrate our work by reference to a worked example, which we outline in this section. The example we describe specifies message streams, routing and rerouting within CCITT's Signalling System No. 7 protocol [**?**] from a number of different ODP viewpoints. The example is based on the specifications of the protocol described in [**?**, **?**], and the informal description is taken from [**?**]. The viewpoints are specified using a combination of Object-Z and LOTOS, and in subsequent sections we show how the correspondences between the viewpoints can be described and how they can be checked for consistency.

The Network Layer of the Signalling System No. 7 protocol provides services which enable a User Part to send message sequences between not necessarily adjacent signalling points in identifiable streams. Messages are confined to streams by routing labels and there are no dependencies across streams within the network. Streams are conceptually concurrent, although an implementation may merge them arbitrarily.

We specify streams using the Computational and Engineering viewpoints. The Computational viewpoint is concerned with the identification of individual correctly sequenced message streams from origin to destination without routing detail, and is specified in Object-Z. The Engineering viewpoint identifies the route of a stream as a sequence of sections, each with a signalling point and an outgoing linkset. This viewpoint is the basis for rerouting. In order to illustrate some of the techniques we have been developing we have further split the engineering viewpoint into two views (i.e. general partial specifications, not necessarily ODP viewpoints): one view which specifies a single stream as a sequence of sections, this is specified in LOTOS; and a second view which describes the multiplicity of these streams, this is specified in Object-Z. These views are at different levels of abstraction and allow us to illustrate how components specified in one viewpoint can be reused in another viewpoint.

We now describe each of these viewpoints in turn, beginning with the Computational viewpoint in Object-Z. Object-Z is an object oriented variant of Z, which extends Z with notions of *class*, *object* and *inheritance*. Object-Z classes that contain no instances of classes in their state can be viewed as encapsulated versions of abstract data types in the standard Z states-and-operations style: they contain a state description, an initialisation, and a collection of operations. Therefore, as a development (refinement) relation for such Object-Z classes we will use standard Z refinement.

Figure 3: Signalling System No. 7

(In particular, in contrast to some of the proposals for Object-Z refinement, we will allow widening of operations' preconditions in refinements. In a context of partial specification this seems the more obvious choice.) For an introduction to Object-Z and LOTOS the reader is referred to [**?**] and [**?**] respectively.

3.1 The Computational Viewpoint in Object-Z

The Computational viewpoint describes service information with respect to streams at the User-Part/Network Layer interface, we do not describe routing or signalling details here.

The set of signalling point codes is denoted SPC, and the codes identify a network's potential signalling points. The SPC of an origin point is denoted *opc*, and that of a destination *dpc*.

$$SPC0..16383$$

There can be up to sixteen streams of communication between any two signalling points. A stream selector is used to distinguish streams between the same two signalling points; these stream selectors are taken from the set SLS.

$$SLS0..15$$

The set of all signalling messages is denoted MSG.

$$[MSG]$$

Each stream is specified as a $STREAM\_ext$ class that describes the state space of the object, its initialisation (in the schema) together with the operations available. For each stream we record the messages actually sent by the application at the origin ($MsgsSent$) and the messages actually delivered ($MsgsDelivered$) to the application at the destination. Two operations define the behaviour of the protocol. The operation *Transmit* accepts a new message and adds it to the *MsgsSent* sequence. The *Receive* operation either causes the output sequence *MsgsDelivered* to increase, or the *MsgsDelivered* sequence is unaltered modelling the environment's busy waiting. Initially, no messages have been sent. (Note that due to the way in which Z models sequences, prefixing is denoted by set inclusion.)

STREAM_ext MsgsSent, MsgsDelivered : MSG MsgsDelivered $\subseteq MsgsSent$
MsgsSent =
Transmit $\Delta(MsgsSent)$
$m? : MSGMsgsSent' = MsgsSentm?$
Receive $\Delta(MsgsDelivered)$
$\#MsgsDelivered' = \#MsgsDelivered + 1 \vee MsgsDelivered' = MsgsDelivered$

The above class describes just one stream, and for each routing label there is a possible stream. A stream is uniquely determined by its origin (*opc*) and destination (*dpc*) point codes together with its stream selector (*sls*). Such an identification is called a routing label (RTG_LAB): RTG_LAB opc, dpc : SPC
sls : SLS The class $STREAM\_EXT$ provides for a multiplicity of streams, and is specified as follows. The operations *GTransmit* and *GReceive* allow the *Transmit* and *Receive* operations to be performed on individual stream objects identified by a particular routing label *lab*?.
STREAM_EXT Stream_ext :RTG_LAB STREAM_ext
$\forall lab : Stream\_ext \dot{S}tream\_ext(lab)$.
GTransmit [lab? : RTG_LAB lab?$\in Stream\_ext]\dot{S}tream\_ext(lab?).Transmit$
$GReceive[lab? : RTG\_LABlab? \in Stream\_ext]\dot{S}tream\_ext(lab?).Receive$

3.2 The Engineering Viewpoint of a single stream in LOTOS

The Engineering viewpoint of a single stream models its path from origin to destination as a chain of sections, one per signalling point en-route, except for the destination point. A section can be considered as an abstraction of a signalling point and its ongoing linkset.

A route is a non-empty sequence of signalling points with no duplicates. The sections join end to end so that the output of one is the input to the next.

The LOTOS specification of an engineering viewpoint stream follows. We assume a given specification *Items* providing the sort *element*.

specification *Sections [transmit,receive]* : **noexit**
type *BBuffer* **is** *Items, Boolean* **with**
  sorts $buffer$
  opns $nil :\rightarrow buffer$
    $add : element, buffer \rightarrow buffer$
    $fst : buffer \rightarrow element$
    $rmv : buffer \rightarrow buffer$
    $empty : buffer \rightarrow Bool$
  eqns
    forall $x, y : element, z : buffer$
    ofsort *element*
      $fst(add(x, nil)) = x;$
      $fst(add(x, add(y, z))) = fst(add(y, z));$
    ofsort $buffer$
      $rmv(nil) = nil;$
      $rmv(add(x, nil)) = nil;$
      $rmv(add(x, add(y, z))) = add(x, rmv(add(y, z)));$
    ofsort *Bool*
      $empty(nil) = true;$
      $empty(add(x, z)) = false;$
endtype
behaviour

hide in, out **in**
$(Section[transmit, out](nil, 0)|||Section[in, receive](nil, succ(0)))$
$|[in, out]|$
$Daemon[in, out]$
where
process $Section[in, out](q : buffer, j : Nat)$: **noexit** :=
$in!j?m : element; Section[in, out](add(m, q), j)$
$[]$
$[not(empty(q))] \rightarrow out!j!(fst(q)); Section[in, out](rmv(q), j)$
endproc
process $Daemon[in, out]$: **noexit** :=
$out?j : Nat?x : element[0 = j]; in!succ(j)!x; Daemon[in, out]$
endproc
endspec

A generic section is described in the process *Section*. This gets messages using action *in*, puts them in a buffer and then, using action *out*, outputs them.

The sections are placed independently in parallel. In order to prevent the example becoming prohibitively complex we only include two sections, but more could be included and furthermore, recursion could be used to accommodate an arbitrary number. For simplicity we use 0 and 1 (written *succ*(0) in LOTOS) as the signalling point codes for the two sections.

A process *Daemon* is composed in parallel with the two sections. This has the role of moving messages between sections.

Further operations to specify any necessary rerouting in the presence of link failure could also be included in the above specification if needed (see for example the rerouting operations given in [**?**]). However, since they do not appear in the computational viewpoint, and are thus not central to consistency checking, we have omitted them in the specification given above.

3.3 The Engineering Viewpoint of multiple streams in Object-Z

The final view(-point) in our example describes how the individual streams are combined to provide a multiplicity of streams. To do so we specify a *STREAM_SECT* class in Object-Z which provides a global picture of the engineering viewpoint in terms of a number of stream objects. Since the stream class was defined in another viewpoint in order to use it here we have to include it in this viewpoint, however, we only define its signature and do not prescribe any behaviour. That is, this viewpoint does not make any assumptions about a stream and the effect of the operations apart from declaring their existence. Consequently, it is guaranteed to be consistent with any viewpoint of a single stream. We will exploit this specification style when we combine the engineering views in the sequel.

Our specification uses the signature of a stream object as defined in LOTOS before. Its signature could be constructed from the Object-Z translation that follows in Section **??**, but also by simply observing which actions occur in the LOTOS specification: *transmit*, *receive*, and internal actions i which we will rename *daemon* to distinguish between the viewpoints. The signature $STREAM\_sect$ contains these operations, with as the only state variables boolean variables representing the applicability of each of the operations.

STREAM_sect predaemon, prereceive, pretransmit : Bool

transmit $\Delta(predaemon, prereceive, pretransmit)pretransmit$

receive $\Delta(predaemon, prereceive, pretransmit)prereceive$
daemon $\Delta(predaemon, prereceive, pretransmit)predaemon$

This will be used to define the class $STREAM\_SECT$. The operations *Gtransmit* and *Greceive* allow the *transmit* and *receive* operations to be performed on individual stream objects identified by a particular routing label *lab*?. The operation *Gdaemon* non-deterministically selects a stream and forwards a message in it from one section to the next by using the *daemon* operation on that stream object.

STREAM_SECT Stream_sect :RTG_LAB STREAM_sect
$\forall lab : Stream\_sect \dot{S}tream\_sect(lab).$
Gtransmit [lab? : RTG_LAB lab?$\in Stream\_sect]\dot{S}tream\_sect(lab?).transmit$
$Greceive[lab? : RTG\_LABlab? \in Stream\_sect]\dot{S}tream\_sect(lab?).receive$
$Gdaemon[\ \exists lab : RTG\_LAB\dot{l}ab \in Stream\_sect]\dot{S}tream\_sect(lab).daemon$

4 Relating LOTOS and Object-Z

Comparing viewpoints written in LOTOS and Object-Z requires that we bridge a gap between completely different specification paradigms. Although both languages can be viewed as dealing with states and behaviour, the emphasis differs between them. Our solution for consistency checking between these two languages is to exploit a behavioural interpretation of Object-Z.

Object-based languages have a natural behavioural interpretation, and there is a strong correlation between classes in object-oriented languages and processes in concurrent systems (see for example [**?**, **?**, **?**]). We have used this correlation as the basis of a translation between the two languages, which has been verified by defining a common semantics for LOTOS and Object-Z.

The ADT component of a LOTOS specification is translated directly into the Object-Z type system. To translate the behavioural aspect of a LOTOS specification we map each 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, the process *Section* in the engineering viewpoint 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.

The translation is given in [**?**], where it is verified against a common semantic model of the two languages. This model is based upon the semantics for Object-Z described in [**?**], which effectively defines a state transition system for each Object-Z specification. This model is used as a common semantic basis by embedding the standard labelled transition system semantics for LOTOS into it in an obvious manner. This provides a basis by which we can verify that the translation is correct, i.e., that the meaning of a term in one language is (bisimulation) equivalent to the meaning of that term after translation. [**?**] verifies the translation in detail.

4.1 Translating the data types

In LOTOS, data types are specified using the language for abstract data types ACT ONE [**?**]. ACT ONE is an algebraic specification method to write parameterised as well as unparameterised ADT specifications. These are translated directly into the Z type system. For example, the specification *Sections* defines a data type given in terms of a signature and a list of equations. The translation of this will introduce a given set to represent the sorts (here $buffer$), together with an axiomatic definition which introduces the operations constrained by the behaviour of the equations. Thus we translate the data type aspect of the specification *Sections* to:

$$[element, buffer]$$

nil : buffer
add : element$\times buffer \rightarrow buffer$
$fst : buffer \rightarrow element$
$rmv : buffer \rightarrow buffer$
$empty : buffer \rightarrow Bool$
$\forall x, y : element, z : buffer \ldots fst(add(x, nil)) = x \ldots fst(add(x, add(y, z))) = fst(add(y, z)) \ldots rmv(nil) = nil \ldots rmv(add(x,$

Moreover, any realistic consistency checking toolbox will also contain direct translations from axiomatic descriptions of standard structured types (e.g. sets, queues and sequences) into their Z mathematical toolbox (cf. [**?**]) equivalents. We will assume that this translation has indeed been made in this example (and hence identify $buffer$ with $element$, $add(x, y)$ with yx, nil with , fst with $head$ and rmv with $tail$).

4.2 Translating the behaviour

The translation of the behaviour of a LOTOS specification produces a number of Object-Z classes, each one representing a behaviour expression (e.g. process definition) of the LOTOS specification. The heart of the translation consists of a number of translation rules, one for each of the LOTOS operators or terminals (i.e. occurrences of *stop*, *exit* or any process instantiations). The translation of a process definition begins with its terminals and successively applies the operator translation rules given in [**?**] until each operator and terminal has been translated.

For example, to translate the behaviour of *Sections*, we first note that it contains two process instantiations *Section*, and another instantiation of *Daemon*. The Object-Z translation will thus contain the definition of the class *Section* and *Daemon* followed by that of *Sections*.

Let us consider the class *Section* first. To translate the behaviour

process $Section[in, out](q : buffer, j : Nat)$: **noexit** :=
$in!j?m : element; Section[in, out](add(m, q), j)$
[]
$[not(empty(q))] \rightarrow out!j!(fst(q)); Section[in, out](rmv(q), j)$
endproc

we begin with the terminals, which in this case are (recursive) process instantiations.

The translation will produce an Object-Z class with state variables q and j and a recursive instantiation to *Section* (in fact we flatten this to a single class here). The operators we have to translate consist of action prefix (;), guarding and choice ([]). The appropriate translation rules are applied

in turn. The events $in!j?m : element$ and $out!j!(fst(q))$ produce two operation schemas, with inputs and outputs to perform the value passing. Each variable declaration, e.g. $?m : element$ gives rise to a state variable of the same name in the Object-Z class. The overall result is the following class:

Section q:buffer; m:element; j:$\mathbb{N}$
in $\Delta(m, q)$
$ch! : \mathbb{N}$
$ch? : element ch! = j \land ch? = m'$
$q' = qm'$
out $\Delta(q)$
$ch_1! : \mathbb{N}$
$ch_2! : element q \neq$
$ch_1! = j \land ch_2! = q$
$q' = q$ (Note that the state variable m could be removed entirely, as its value is never used when it does not equal $ch?$. This need not be the case in general for variables containing received values, though.)

An interesting observation, illustrated by this example, is that translation from LOTOS to Object-Z will always result in classes with no invariants defined on the state components. However, invariants have a valuable role in consistency checking, as they allow unreachable states to be eliminated from consideration. Thus, as a preliminary stage of consistency checking, often class invariants (that are established by initialisation and preserved by all operations) are made explicit.

The translation of the *Daemon* process follows similar lines, to produce the class:

Daemon x :element
j :$\mathbb{N}$
s : {0,1}
s = 0 in $\Delta(s)$
$ch_1! : \mathbb{N}$
$ch_2! : elements = 1 \land s' = 0$
$ch_1! = j + 1 \land ch_2! = x$
out $\Delta(s, j, x)$
$ch_1? : \mathbb{N}$
$ch_2? : elements = 0 \land s' = 1$
$ch_1? = j' = 0 \land ch_2? = x'$

Finally, we translate the behaviour

hide in, out **in**$(Section[transmit, out](nil, 0)|||Section[in, receive](nil, succ(0)))|[in, out]|Daemon[in, out]$

The process instantiation rule produces a class with two instances of the object *Section* and one instance of the object *Daemon*. Subsequently we need to translate the parallel compositions induced by $|||$ and $|[in, out]|$. The former interleaves the operations defined in the two instances of *Section*, if these are denoted s_1 and s_2 then this interleaving results in operation definitions of the form $transmits_1.in$ etc. The synchronisation with *Daemon* induced by $|[in, out]|$ produces a similar synchronisation in the Object-Z class using the Object-Z parallel operator $||$ to produce operations of the form $s_2.in||d.in$, where d is an object of type *Daemon*. Finally, hiding in and out produces an internal operation. The complete specification of the class *Sections* can then be given as

Sections $s_1, s_2 : Section$
$d : Daemon$
$s_1. \land s_2. \land d.$
transmit $s_1.in$

$receives_2.out$
$i(s_2.in||d.in) \vee (s_1.out||d.out)$

In order to illustrate the consistency checking techniques in a convenient way, we will in fact flatten this specification into a specification consisting of just one class *Sections*. This is easily achieved by including the definitions of *Section* and *Daemon*, indexing state variables to differentiate between the different instances of *Section*. If we then also remove constant and irrelevant components, we end up with the single class: Sections $\mathrm{q}_1, q_2 : buffer$
$x : element$
$s : \{0, 1\}$
$\mathrm{q}_1 = q_2 = \wedge s = 0$
transmit $\Delta(q_1)$
$ch! : \mathbb{N}$
$ch? : elementch! = 0$
$q_1' = q_1 ch?$ receive $\Delta(q_2)$
$ch_1! : \mathbb{N}$
$ch_2! : elementq_2 \neq$
$ch_1! = 1 \wedge ch_2! = q_2$
$q_2' = q_2$
i (in —— din) $\vee(out||dout)$
$in\Delta(q_2)$
$ch_1! : \mathbb{N}$
$ch_2? : elementch_1! = 1$
$q_2' = q_2 ch_2? din\Delta(s)$
$ch_1! : \mathbb{N}$
$ch_2! : elements = 1 \wedge s' = 0$
$ch_1! = 1 \wedge ch_2! = x$
$out\Delta(q_1)$
$ch_1! : \mathbb{N}$
$ch_2! : elementq_1 \neq$
$ch_1! = 0 \wedge ch_2! = q_1$
$q_1' = q_1 dout\Delta(s, x)$
$ch_1? : \mathbb{N}$
$ch_2? : elements = 0 \wedge s' = 1$
$ch_1? = 0 \wedge ch_2? = x'$

5 Correspondences

In order to establish whether these viewpoints are consistent, we first need to describe formally how they are related. In particular, clearly they overlap in parts of the envisaged system that they describe (e.g. all the viewpoints above specify the result of receiving a message), but this needs to be documented formally. Thus, we have to establish the *correspondences* between the viewpoints.

What are the correspondences in the above example? There are (at least) three possible correspondences between the viewpoints, and these are illustrated in the following diagram (the correspondences are marked **a**, **b** and **c**). Each correspondence relates terms (e.g. names of classes, operations, state variables) in two viewpoints and the correspondences in this example illustrate an increasing complexity according to how tightly the viewpoints are coupled.

The simplest correspondence, **a**, links the two engineering views and simply identifies the *Sections* class/process with its use as a component in the multiple stream view. This is described by saying that, for example, *daemon* and i represent different perspectives of the same event and so we

Figure 4: Correspondences between viewpoints

should link them. Let us document these correspondences in a table.

EngVpt multiple	EngVpt single	Status
$STREAM\_sect$	$Sections$	**unify**
$transmit$	$transmit$	**unify**
$receive$	$receive$	**unify**
$daemon$	i	**unify**
$predaemon, prereceive, pretransmit$	q_1, q_2, x, s	**relate**

The following "Status" labels will be used:

unify The behaviours are intended as describing aspects of the same system part, they are not necessarily equal but should be checked for consistency.

equal The corresponding items are intended to describe exactly the same system part in exactly the same way.

relate The corresponding state components represent related information – a predicate should be given to describe how they relate.

The last column in the table above illustrates that when the Object-Z classes themselves have a **unify** label, this implies that their respective states need to be related. Given that the booleans which make up the state of the $STREAM\_sect$ signature class represent applicability of the operations, it seems obvious to relate them to the preconditions of the corresponding operations in $Sections$. It is often convenient to represent **relate** predicates as schemas, in this case: EngVpt-tRelate predaemon, prereceive, pretransmit: Bool
$q_1, q_2 : buffer$

$x : element$
$s : \{0, 1\} predaemon \Leftrightarrow i$
$pretransmit \Leftrightarrow transmit$
$prereceive \Leftrightarrow receive$ The correspondence **b** between the computational viewpoint and the single stream engineering view links *STREAM_ext* to *Sections*, and is much more involved. Clearly the protocol transmits one type of message, so MSG and *element* should be identified. Again the operations and actions described in the two viewpoints are different perspectives of the same function, so we should link $Transmit$ to *transmit* and *Receive* to *receive* (and implicitly the input $m?$ of $Transmit$ is identified with the input of *transmit*). Finally, it is clear that $MsgsSent$ and $MsgsDelivered$ in the computational viewpoint in some way represent information part of which is also represented by the buffers q_1 and q_2 in the engineering viewpoint. However, a crucial difference between the viewpoints is that *Receive* has no output, whereas *receive* does. Actually, the values output (in $ch_2!$) by *receive* together constitute $MsgsDelivered$. This is no unsurmountable problem – in terms of refinement a specification with outputs and one which contains, in addition, an accumulated sequence of these outputs, are equivalent.[2] Thus, in order to completely exhibit the correspondences between the viewpoints, we add a sequence *delivered* to *Sections*, which is initialized to and only changed in *receive*, as follows: receive $\Delta(q_2, delivered)$
$ch_1! : \mathbb{N}$
$ch_2! : elementq_2 \neq$
$ch_1! = 1 \wedge ch_2! = q_2$
$q_2' = q_2$
$delivered' = deliveredch_2!$ Then we have $MsgsDelivered = delivered$. (Note that this does not impact on the correspondence between the engineering viewpoints.) $MsgsSent$ consists of many parts: all the messages queued in q_1, possibly a message in transit between the two internal operations, all the messages queued in q_2 and all the messages delivered. The correspondences can now be given in the form of a schema as follows: EngsCompRelate $q_1, q_2, MsgsSent, MsgsDelivered, delivered : buffer$
$x : element$
$s : \{0, 1\} MsgsDelivered = delivered$
$s = 0 \Rightarrow MsgsSent = deliveredq_2q_1$
$s = 1 \Rightarrow MsgsSent = deliveredq_2xq_1$Again, we can document these correspondences as a table

CompVpt	EngVpt single	Status
$STREAM\_ext$	*Sections*	**unify**
MSG	*element*	**equal**
$Transmit$	*transmit*	**unify**
Receive	*receive*	**unify**
delivered	$MsgsDelivered$	**equal**
$q_1, q_2, x, delivered$	$MsgsSent$	**relate**

The final correspondence **c** between the computational viewpoint and the multiple stream engineering view is the most complex as both viewpoints provide an internal representation of the same class. In particular, they both define streams between routing labels, and the streams and the routing labels used must be related. Clearly the same set of routing labels has to be used in both viewpoints, so they are identified by saying that the domains of the indexing functions are identical (i.e. $Stream\_ext = Stream\_sect$). The second constraint is that for a given routing label ($lab : Stream\_ext$) the computational and engineering stream objects are the same stream (i.e. these really are just different perspectives of the *same* object) related by the correspondence **b**.

The correspondence **c** is now written as a schema: EXT_SECT STREAM_EXT

[2]Technical details on this can be found in [**?**], section 16.5 on "unwinding", and in [**?**], the discussion on non-trivial finalisations. These methods, or the IO-refinement method discussed in [**?**] can also be used to show that the constant outputs $ch_1!$ are irrelevant, and that even the specification which contains the sequence *instead of* the outputs is equivalent.

STREAM_SECT Stream_ext = Stream_sect

$\forall lab : Stream\_ext \bullet \exists STREAM\_ext; STREAM\_sect; Sections \bullet STREAM\_ext = Stream\_ext(lab) \bullet STREAM\_sect =$

Note that the existentially quantified *Sections* means that there is an instance of this viewpoint for each *lab*, but unlike for the other viewpoints, these instances are not gathered into an indexed collection.[3]

6 Consistency in Object-Z and Z

6.1 Overview

We have now obtained viewpoints which are all specified in Object-Z. Moreover, they are all in the subset of Object-Z that is naturally viewed as encapsulated standard Z states-and-operations specifications, viz. classes that have no objects in their states. For Object-Z classes of this shape, we can adapt the consistency checking techniques for Z described in [**?**]. Recall that consistency is defined as the existence of a common refinement (unification). In Z refinement [**?**, **?**], we allow reduction of non-determinism (strengthening of postconditions) and extension of the domain (weakening preconditions) of operations. Additionally, *data* refinement is possible.

The Z unification techniques in [**?**] operate on two viewpoints at the same level of decomposition, however here we have three viewpoints, two of which describe collections of "stream" objects based on single stream descriptions, and one of which only describes a single stream. Fortunately, the way in which the "multiple streams" descriptions have been based on their respective single stream descriptions ensures that a consistency check of the three single stream descriptions, when successful, guarantees consistency of the complete specifications. The construction mechanism used in $STREAM\_EXT$ and $STREAM\_SECT$ is called *free promotion*, and a theorem in [**?**] states that the promotion of a refinement is a refinement of the promotion, provided the promotion is free. Thus, a (least) common refinement of the single stream descriptions, when promoted to multiple streams, is a common refinement of the full viewpoints, i.e. a witness to their consistency.

Therefore we will initially concentrate on the three single stream descriptions, construct a unification of those, and then promote it to a witness of consistency of all three viewpoints.

In addition, some of the viewpoints given here have internal operations, so the refinement relation we need to use is actually *weak refinement* [**?**]. However, the consequences of this for the consistency checking process are only minor, and will be highlighted where they occur.

A unification of two viewpoints is constructed in two phases. In the first phase ("state unification"), a unified state space (i.e., a state schema) for the viewpoints has to be constructed. The essential components of this unified state space are the correspondences between the types in the viewpoint state spaces. At this stage we have to check that a condition called *state consistency* is satisfied. The viewpoint operations are then adapted to operate on this unified state.

In the second phase, called *operation unification*, pairs of adapted operations from the viewpoints which are linked by a correspondence (e.g. *Transmit* and *transmit*) 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. A similar procedure also needs to be executed for the initialisations of the viewpoints, and the adapted initialisations together need to be satisfiable.

[3] Actually, only the existential quantification over *Sections* is a genuine one – the other two are artefacts of the Z schema notation, allowing the direct inclusion of $EngVptRelate$ etc. without explicit substitutions.

6.2 State Unification

The first step in state unification is to establish whether the *state consistency condition* is fulfilled. If the viewpoint state schemas are $\mathrm{S}_1 x : Spred_1$ $\mathrm{S}_2 y : Tpred_2$ and their correspondence is given as R x:S; y:T pred_R then state consistency is

$$\forall x : S; y : T \dot{p}red_R \Rightarrow (pred_1 \Leftrightarrow pred_2)$$

i.e., no legal state of one viewpoint is linked to an illegal one of the other.

A natural way of avoiding state inconsistencies is to include the entire state schemas in the correspondence schema, i.e. to have a correspondence of the form R' $\mathrm{S}_1; S_2 pred_{R'}$ which ensures that both viewpoint predicates hold for all values in the correspondence schema.

Even though we will construct the unification of all viewpoints by pairwise unifications and promotion, we can analyze the correspondences for state consistency together:

- The correspondence schema $EngVptRelate$ introduces no state inconsistencies as the viewpoint states involved have empty predicate parts.
- For $EngsCompRelate$, we need to prove that its predicate implies the state predicate of $STREAM\_ext$, which is that $MsgsDelivered \subseteq MsgsSent$. This is trivial, as $EngsCompRelate$ states that $MsgsDelivered = delivered$ and $MsgsSent$ consists of a concatenation of sequences, the first of which is $delivered$.
- For EXT_SECT, state consistency is trivial because the correspondence schema includes the state schemas $STREAM\_EXT$ and $STREAM\_SECT$.

The next phase of state unification is the totalisation of correspondence relations. In brief, to get the most general refinement possible, every value allowed in either of the viewpoint states needs to be represented in the unification – even if it is not linked to any value by the correspondence. If the correspondence is already total, we can use the correspondence schema itself as the unified state. In that case, operations Op will be adapted to[4] ΔROp. In this example, all correspondences are total:

- $EngVptRelate$ is total: for every instance of $Sections$ one can find corresponding booleans $prereceive$ and $predaemon$; vice versa, in $Sections$ none, either, or both of $receive$ and i can be enabled at any time.
- $EngsCompRelate$ is also total. In one direction that is trivial, as the $STREAM\_ext$ components are defined as expressions in terms of the $Sections$ components. In the other direction, whenever $MsgsDelivered \subseteq MsgsSent$, $MsgsSent$ equals $MsgsDeliveredq_1 q_2$ for some q_1, q_2.
- Finally, the correspondence EXT_SECT is total as well: for every $STREAM\_EXT$ there is a $STREAM\_SECT$ such that their state components (functions) have the same domains, and such that their respective images match, and vice versa.

6.3 Unifying the engineering viewpoints

As discussed above, we will first construct a unification for the three "single stream" descriptions. This is done via two binary unifications, starting with the simplest one. $STREAM\_sect$ has been

[4]Object-Z operations can be specified in the form $\Delta(c_1, \ldots, c_n)pred$, leaving implicit that some components remain unchanged. Before adaptation Op should be rewritten in a form $\Delta Spred'$, such that S is the complete state.

constructed as a "signature" of *Sections*, it does not constrain the behaviour of *Sections* at all, and only models that some of its operations are partial. Thus, we would expect their unification to proceed smoothly.

In analyzing the correspondences above, we already found that the correspondence is total, and can be used as the common state for their unification. However, as the boolean components can be derived directly from the other state components, we will take the equivalent but simpler state of *Sections*. Any occurrence of *prereceive* and *predaemon* should then be replaced by *receive* and i, respectively.

The operations from $STREAM\_sect$, when adapted to the common state, become:

1. $Sstransmit \Delta Sections$
2. $Ssreceive \Delta Sectionsreceive$
3. $Ssdaemon \Delta Sectionsi$

The initialisation of $STREAM\_sect$ becomes *Sections*. The operations from *Sections* do not need to be adapted as they already operate on the correct state, and the correspondence is total.

The rule for operation unification is as follows [**?**, **?**]. Two operations $Op1$ and $Op2$, both changing state S and with input $x?1{:}1T$, are unified to

OpΔS
$x?1{:}1TOp11 \vee 1Op2$
$Op111Op1$
$Op211Op2$For this unified operation to be a common refinement of the original operations, the condition of *operation consistency* needs to hold: whenever both pre-conditions hold, $Op1 \wedge Op2$ must be satisfiable. This clearly represents the informal notion that the two viewpoint operations should not impose contradictory requirements. Additionally, if the operations concerned are *internal* operations, we need to ensure that their preconditions *coincide*. This follows from the conditions for weak refinement in [**?**].

According to the table in section **??**, the following pairs of operations need to be unified: $Sstransmit$ and *transmit*; $Ssreceive$ and *receive*; $Ssdaemon$ and i. All of these are trivial, resulting in *transmit*, *receive* and i. For example, the last is: uni$\Delta Sections Ssdaemon1 \vee 1i$
$Ssdaemon11Ssdaemon$
$i11i$ which indeed equals i, and satisfies the operation consistency condition, given that $Sssdaemon = i$. The extra weak refinement condition for internal operations, viz. equality of preconditions, is clearly also fulfilled. The two initialisations are also consistent.

In conclusion, the unification of *Sections* and $STREAM\_sect$ is *Sections* itself, as we would have hoped. A clear advantage of this is that, in order to establish a three-way unification in the next step, we do not have to compose correspondence relations to end up with one relating the third with the unified first two viewpoints. Instead, the correspondence between *Sections* and $STREAM\_ext$ will suffice.

6.4 Unifying the computational and engineering viewpoints

When unifying the computational and engineering viewpoints, we need to adapt their operations to the unified state given by their correspondence schema (section **??**), viz. EngsCompRelate
$q_1, q_2, MsgsSent, MsgsDelivered, delivered : buffer$
$x : element$

$s : \{0, 1\} MsgsDelivered = delivered$
$s = 0 \Rightarrow MsgsSent = deliveredq_2q_1$
$s = 1 \Rightarrow MsgsSent = deliveredq_2xq_1$and then perform operation unification on the corresponding pairs of operations. We do this in turn for the initialisation and each operation.

6.4.1 Initialisation

The computational viewpoint initialisation, when adapted to the unified state, is CInit EngsCompRelate MsgsSent = which requires all the $buffer$ components to be empty, and s to be 0. The engineering viewpoint initialisation is EInit EngsCompRelate $q_1 = q_2 = delivered =$
$s = 0$ which is satisfiable and equivalent to $CInit$ (and thus consistent).

Thus, the initialisation in the unification will be either of the two equivalent adapted initialisations.

6.4.2 Transmission

First, $Transmit$ is adapted to $adTransmit\Delta EngsCompRelateTransmit$, which is after simplification: adTransmit $\Delta(MsgsSent, q_1, q_2, x, s)$
$m? : elements = 0 \Rightarrow MsgsSent = deliveredq_2q_1$
$s = 1 \Rightarrow MsgsSent = deliveredq_2xq_1$
$s' = 0 \Rightarrow MsgsSent' = deliveredq_2'q_1'$
$s' = 1 \Rightarrow MsgsSent' = deliveredq_2'x'q_1'$
$MsgsSent' = MsgsSentm?$ Informally, the new value $m?$ needs to be added to the end of $MsgsSent$, but not necessarily in the obvious way by adding it to q_1; for example if q_1 is empty it may be put in x or q_2.

The corresponding operation in the engineering view is $transmit$ which gets adapted to $adtransmit\Delta EngsCompRelat$ which is after simplification and identification of input $ch?$ with $m?$: adtransmit $\Delta(MsgsSent, q_1)$
$m? : elementq_1' = q_1m?$
$MsgsSent' = MsgsSentm?$ Clearly $adtransmit$ is a refinement of $adTransmit$, and thus the former is the least common refinement (operation unification) of both. In conclusion, for this operation the engineering viewpoint specialises the computational viewpoint.

6.4.3 Reception

The $receive$ operation in the engineering viewpoint gets adapted to $adreceive\Delta EngsCompRelatereceive$, which simplifies to adreceive $\Delta(q_2, delivered, MsgsDelivered)$
$ch_1! : \mathbb{N}; ch_2! : elementdelivered' = deliveredch_2!$
$ch_1! = 1$
$ch_2! = q_2$
$q_2 = q_2$
$MsgsDelivered' = MsgsDeliveredch_2!$ The precondition of this operation is that q_2 is nonempty.

The $Receive$ operation in the computational viewpoint is a disjunction of two operations, we will consider each of those separately. The first is total, models active waiting, and gets adapted to adWait $\Delta EngsCompRelateMsgsDelivered' = MsgsDelivered$
$MsgsSent' = MsgsSent$ The second models successful reception of a value, and gets adapted to adSReceive $\Delta EngsCompRelate\#MsgsDelivered' = \#MsgsDelivered + 1$
$MsgsSent' = MsgsSent$ In terms of the other components, $\#MsgsDelivered' = \#MsgsDelivered+$ 1 becomes $MsgsDelivered' = MsgsDelivered(q_2q_1)$ when $s = 0$ (and similar when $s = 1$). Thus, the precondition of $adSReceive$ is adSReceive EngsCompRelate (s=0 $\wedge\ q_2q_1 \neq) \vee$

$(s = 1 \ \wedge \ q_2 x q_1 \neq)$
which is slightly weaker than the precondition of *adreceive*.

The operation unification of *adreceive* and $adWait \vee adSReceive$ is[5] unReceive $\Delta EngsCompRelate$
$ch_1! : \mathbb{N}; ch_2! : elementadreceive \vee (adWait \vee adSReceive)$
$adreceive \Rightarrow adreceive$
$(adWait \vee adSReceive) \Rightarrow (adWait \vee adSReceive)$ which simplifies to $(adWait =)$ unReceive $\Delta EngsCompRelate$
$ch_1! : \mathbb{N}; ch_2! : elementq_2 \neq\Rightarrow adreceive$
$adWait \vee adSReceive$ Also, the operations are consistent: when both preconditions hold, i.e. when q_2 is nonempty, both allow *adreceive* to happen. The unified operation in this case is determined by both viewpoints: within the engineering viewpoint operation's domain, the computational viewpoint's behaviour is made more deterministic; the latter however provides a wider precondition.

6.4.4 Internal operation

Apart from the normal operations that occur in both viewpoints, there is also an internal operation that occurs only in the engineering viewpoint. The consistency checking rules for standard Z refinement do not suffice here. Informally, for consistency one would expect internal operations in one viewpoint to have no noticeable effect on the state in the other. Formally, this does indeed follow from the rules for weak refinement in [**?**]. Operation consistency for an internal operation in only one viewpoint means it should be unified with an identity operation on the other. Additionally, because the precondition of an internal operation may not be weakened in weak refinement, this identity operation should be partial, its precondition corresponding to the internal operation's precondition. Finally, one needs to ensure that no divergence is introduced in refinement, i.e. no internal operation should be infinitely often enabled before or after a normal operation happens.

The internal operation in the engineering viewpoint is a disjunction of two operations $in||din$ and $out||dout$, we will give their adaptations separately.

First, $adin\Delta EngsCompRelatein||din$ simplifies to adin $\Delta(q_2, s)q_2' = q_2 x$
$s = 1 \wedge s' = 0$ The operation $adout\Delta EngsCompRelateout||dout$ simplifies to adout $\Delta(q_1, s, x)q_1 \neq$
$x' = q_1$
$q_1' = q_1$
$s = 0 \wedge s' = 1$ Operation unification of these with partial identities on the $STREAM\_ext$ state means that we need to prove that both these operations allow the sequences $MsgsDelivered$ and $MsgsSent$ to remain unchanged when these operations are applied. We have actually proved something stronger: those sequences do not appear in the Δ lists, so they are *required* to remain unchanged. Thus, the operation unifications we were looking for are these operations themselves. The operation $adin \vee adout$, an internal operation, is thus part of the unified viewpoint. One can establish that it is non-divergent as required: it transfers messages from q_1 to q_2 in two steps, which can only happen a finite number of times since q_1 will have a finite length only.

6.5 Conclusion

We have now established the consistency of the three single stream descriptions and constructed their unification to witness this. It is given by a class whose state is the overall correspondence *EngsCompRelate*, and whose initialisation and operations are the results of operation unification

[5]Formally, to allow introduction of outputs in refinement, which *Receive* did not have, we would need IO-refinement or similar rules, as mentioned before in section **??**.

Figure 5: LOTOS Consistency Relations

given above, i.e.: StreamU EngsCompRelate CInit adtransmit
unReceive i adin $\vee adout$ which is a (least common) refinement of each of the viewpoint classes. As a consequence, the promoted version of this, viz. STREAMU Streamu: RTG_LAB StreamU
$\forall lab : Streamu \dot{S}treamu(lab)$.
UTransmit [lab? : RTG_LAB lab?$\in Streamu]\dot{S}treamu(lab?).adtransmit$
$URecieve[lab? : RTG\_LABlab? \in Streamu]\dot{S}treamu(lab?).unReceive$
$Udaemon[\ \exists lab : RTG\_LAB\dot{l}ab \in Streamu]\dot{S}treamu(lab).i$ is a refinement of $STREAM\_SECT$ and $STREAM\_EXT$, and thus a witness of consistency of the entire collection of viewpoints.

This concludes the proof of consistency of the viewpoints. Once the correspondences had been set up as required (with the issue of "remembering" outputs as the main hurdle), previously published techniques [**?**] allowed the proof to proceed. In some cases we needed to use generalised refinement conditions (weak and IO-refinement [**?**, **?**, **?**]), but these had only local effects. The main effort was in simplification of the resulting schemas, and in knowing when to tacitly use distributivity of disjunction over refinement.

Automation of this consistency proof would have been possible to a large extent. The simplifications of schemas were useful in increasing our confidence in the correctness of the result – any automated consistency check would need user guidance to come up with the "right" simplifications. Also, the way the use of the generalised refinement rules could be relegated to side comments suggests that these rules should only optionally be used in any automated system. One would rarely want to deal with the full generality of the rules in [**?**, **?**] indeed.

7 Consistency in LOTOS

In addition to the techniques discussed so far, we have also developed mechanisms to check the consistency of two viewpoint specifications written in LOTOS. This section reviews our work in this area.

Instantiations of Consistency. A major influence on consistency in LOTOS is that the language supports a large spectrum of development relations. Elsewhere we have categorised consistency according to these different relations [**?**, **?**, **?**], which is summarised in figure **??**. The development relations highlighted are the following:

$\leq_{tr}$ - *trace preorder (i.e. refinement as preservation of safety properties); ext - extension (i.e.*

refinement as addition of functionality); conf - conformance; red - reduction (i.e. refinement as reduction of non-determinism); cs = conf∩ $conf^{-1}$; xcs = ext∩$conf^{-1}$; te - testing equivalence; ≈ - weak bisimulation equivalence; ∼ - strong bisimulation equivalence.

The figure considers instantiations of consistency with each of these development relations, e.g. C_{red} denotes consistency when the development relation is instantiated as *red*. The figure illustrates, as a Venn diagram, the relative strengths of the different instantiations of consistency. For example, it indicates that $C_{\sim}$, consistency according to strong bisimulation, is the most discriminating check. In other words, if two specifications are consistent by $C_{\sim}$ they will be consistent by all other instantiations of consistency, however, there is at least one pair of specifications that is consistent by all other instantiations, but not by $C_{\sim}$. At the other extreme, the instantiations $C_{\leq_{tr}}$, C_{ext} and C_{conf} are completely un-discriminating, in the sense that all pairs of LOTOS specifications are consistent according to these checks.

A full discussion of the different development relations we have considered and the resulting notions of consistency is beyond the scope of this paper. However, a general point should be clear, which is that there are many different notions of consistency all arising from different notions of development, and these can be related according to their relative strength. This enables appropriate consistency checks to be employed according to the class of viewpoint specification being considered.

Example. We illustrate the LOTOS consistency checking techniques using our running example. To contain the complexity of our presentation, the illustration is slightly artificial, but it will serve to highlight our approach.

The LOTOS process *Section* presented in section **??** implements a buffer which inputs data items using the action *in*, adds them to a queue and then retrieves items from the queue using action *out*. Such a behaviour can be viewed as a unification of two partial specifications. The first is a "lossy" section. It is defined as follows:

process $SectionPS1[in, out](q : buffer)$: **noexit** :=
  $in?m : element; SectionPS1[in, out](add(m, q))$
  []
  $in?m : element; SectionPS1[in, out](q)$
  []
  $[not(empty(q))] \rightarrow out!fst(q); SectionPS1[in, out](rmv(q))$
endproc

which adds the option, the second branch of the three way choice, to lose the item that is input. Notice that a non-deterministic choice on *in* (for a particular data item) results between the first branch and the second branch. Also, we have not included the section identifier, the j, which was in the original version. This is for simplicity of presentation; it could easily be added without affecting our approach. The ADT definition presented in section **??** with *Sections* is assumed to be generic and hence available to all the partial specifications we consider here.

We also assume the following partial specification:

process $SectionPS2[in, out](q : buffer)$: **noexit** :=
  $in?m : element; SectionPS2[in, out](add(m, q))$
  []
  $in?m : element; SectionPS2[in, out](add(m, add(m, q)))$
  []
  $[not(empty(q))] \rightarrow out!fst(q); SectionPS2[in, out](rmv(q))$
endproc

which models a section which on inputting a data item has the (non-deterministic) option to duplicate the data item, modelled by adding the item twice to the queue.

Here we assume that the correspondences between $SectionPS1$ and $SectionPS2$ are given implicitly, by name. With this in mind we can see that the behaviour of each partial specification constrains the behaviour of the other specification. $SectionPS2$ constrains the behaviour of $SectionPS1$ by not allowing the option to lose data items, while $SectionPS1$ constrains the behaviour of $SectionPS2$ by not allowing the option to duplicate data items. Consequently when we unify the two specifications using reduction we obtain $Section$, which only exhibits the common behaviour.

Research based on work performed by Leduc [**?**] can be used to characterise the least developed unification according to reduction. Specifically, if we denote a least developed unification of two processes P and Q by U then the following trace/refusal property characterises U (the reader unfamiliar with trace/refusals is referred to [**?**]):

$$Tr(U) = Tr(P) \cap Tr(Q) \quad \wedge \quad \forall \sigma \in Tr(U) \,.\, Ref(U, \sigma) = Ref(P, \sigma) \cap Ref(Q, \sigma)$$

It turns out that if we replace U by $Section$, P by $SectionPS1$ and Q by $SectionPS2$, then the above relationship holds.

Unbalanced Unification. What we have presented is a very simple example which illustrates one of the simplest of our unification strategies: unification according to reduction. However as indicated earlier in this section, there are a large number of different development relations associated with LOTOS. Using the work of Guy Leduc [**?**] as a starting point we have identified unification algorithms for all the major combinations of LOTOS development relations [**?**]. In particular, we have considered the important issue of unbalanced consistency - where different viewpoints are related to the unification by different development relations. This is a common situation with viewpoints modelling.

For example, as a rather contrived illustration, we could consider a process:

process $SectionPS3[in, out](q : buffer)$: **noexit** :=
  $in?m : element; SectionPS3[in, out](add(m, q))$
endproc

which just allows items to be added to the section, but never offers the action out. Then we might require that our unification is a common reduction of $SectionPS1$ and $SectionPS2$, but is also an extension of $SectionPS3$. Extension enables new behaviour to be added to a partial specification as long as new deadlocks are not added. $Section$ would indeed be a suitable (3 way) unification.

We have characterised consistency and unification for a spectrum of such unbalanced situations. These are reported in [**?**, **?**].

8 Issues and Tool Support

The key component of the consistency checking strategy presented here is to be able to identify common refinements of multiple viewpoints with respect to the correspondences between the viewpoints. Such refinements can also be viewed as common *models* for the collection of viewpoints. These common models will typically be expressed in terms of the most primitive entities in the viewpoints, for example in the protocol viewpoints typical entities included: actions or operations, e.g. $transmit$ and $Receive$; data variables, e.g. the sequence $MsgsDelivered$ representing messages delivered by the protocol.

However, finding a suitable set of primitives is not always possible. In particular, different ODP viewpoints occur at different levels of abstraction, thus identifying one-to-one correspondences is almost certain to be impossible in general. In fact, these correspondences can be extremely complex with what are primitive entities in one viewpoint being related to whole portions of behaviour in another viewpoint. For example, the execution of a remote procedure call operation in the computational viewpoint would actually correspond to a body of primitive interactions in the engineering viewpoint, e.g. interactions between stub objects, binding objects and protocol objects in order to invoke an RPC transport protocol.

This difficulty raises many questions about how the viewpoints are specified, how to document the correspondences and how to deal with changes in the level of abstraction between viewpoints. General viewpoint models have great difficulty dealing with correspondences, having to use similarity checking [**?**] or low level common models [**?**]. However, because ODP has a fixed set of viewpoints with predetermined roles, more specific guidance can be provided for establishing correspondences.

Describing correspondences

The ODP reference model prescribes a number of correspondences between particular viewpoints (e.g. between computational objects and basic engineering objects). Under current practice common terms are identified by name alone [**?**]. However, in general, between viewpoints at different levels of granularity, we need to relate single actions and objects to complete behaviours. To do this the viewpoint specifications need to be structured with correspondences in mind and in appropriate ways. It is also becoming increasingly clear that consistency checking will only be feasible if the correspondences between viewpoints are considered at the same time as the viewpoints themselves are being structured as opposed to attempting to retrofit inappropriate correspondences later. For example, in the case study we needed to remedy the situation that one viewpoint "remembered" past outputs but the other did not. As a consequence, one viewpoint specification had to be changed when the correspondence was being established. (Fortunately in this case the change made no difference to the semantics of the specification, so we were at liberty to do so.)

There has been work on structuring complex specifications for single viewpoints, e.g. work on templates for ODP [**?**] and specification architectures [**?**]. However, to date there has been little work on exploiting the facilities provided by specific languages which allow viewpoints to be related and combined by elegant structuring mechanisms. To do so specifications at different levels of granularity need to be related.

The nature of ODP viewpoints can be exploited here by providing techniques that reflect their relationships. For example, the engineering viewpoint may provide standard communication components that are assumed when describing a computational viewpoint specification. This needs to be enhanced with mechanisms to relate portions of behaviour between viewpoints, for example by using notions of action refinement.

Changing granularity

Action refinement incorporates a change of action granularity into the refinement. It fits naturally into a process algebra setting where actions serve as the primitive unit of computation. For example, in the engineering viewpoint written in LOTOS one branch dealt with reception of a message[6]:

$$in; Section[in, out](add(m, q), j)$$

[6]For simplicity, we have removed the output here.

and we may wish to refine this to show how the message is passed down the layers in the protocol stack:

$$inlyr_1; \ldots; inlyr_n; Section[in, out](add(m, q), j)$$

where the action *in* has been action refined into the "partial behaviour" $inlyr_1; \ldots; inlyr_n$.

The first behaviour could be viewed as more "abstract" in its modelling of the transmission process; the actual mechanism for communication is abstracted away from and represented by a single action. This method of action refinement enables us to relate viewpoints at different levels of abstraction to the same unification. For example, one viewpoint, expressed in terms of coarse grain primitives, could be action refined to a model that is expressed in terms of the finer grained primitives of another viewpoint.

Such action refinement has been quite extensively investigated within the process algebra field, although little work has to date been performed in the context of LOTOS. However, there are some underlying problems with action refinement. In particular, it has been realised that it is difficult to handle in the context of an interleaving semantics (which is the standard approach), because central to interleaving semantics is the assumption that actions are atomic. Clearly, if actions can be refined into arbitrarily complex behaviours, it is hard to sustain the assumption of atomic actions. Research has suggested that true concurrency models are better behaved in the presence of action refinement [**?**], since true concurrency models do not rely on the assumption of atomic actions.

A different method of providing support for a change of granularity was already illustrated in the example presented above. This was the use of a single stream component in the multiple stream engineering view. We exploited here the use of *promotion* in Object-Z when we promoted the operations defined in the skeleton *STREAM_sect* class to an operation in the *STREAM_SECT* class. To perform this promotion (a similar facility exists in Z as well) all we needed to know was the signature of the component. The behaviour of the component was defined in a separate viewpoint and the correspondence relation was trivial (it just linked up names). The advantage of this style is that it automatically guarantees the consistency of the two engineering views, and to unify them all that is needed is the renaming of the signatures as specified in the correspondence.

Other viewpoints

The work described above has considered consistency checking in LOTOS and Object-Z/Z, this has been successful as proof of concept, however to have a practical impact this work needs to extended to other notations and particular viewpoints. Integration with viewpoints written in UML and support for the enterprise viewpoint are explored in [**?**, **?**]. Although the example specified above describes a protocol, none of the viewpoints presented so far actually guarantee delivery of the messages, and one could imagine an enterprise viewpoint which specifies that eventually messages are delivered. This could be achieved by use of a temporal logic to specify that

$$\forall n : \mathbb{N} \Box (\#MsgsSent = n \Rightarrow \Diamond \#MsgsDelivered = n)$$

that is, it is always the case that eventually the sequence $MsgsDelivered$ will have the same length that $MsgSent$ had before, i.e. we will eventually deliver every message. The relation to other viewpoints could then perhaps be maintained by embedding this temporal specification as part of a history invariant in an Object-Z class. (History invariants are temporal logic statements expressed as part of an Object-Z specification [**?**].)

A further avenue of investigation relates to the use of different languages in different viewpoints. Each language (whether informal or formal) has an associated development relation, or in the case of some languages such as LOTOS, more than one development relation. If the viewpoints

are to be developed separately according to different development or refinement relations, the relationship between different notions of refinement need to be documented.

To this extent we have considered how the refinement relations in Z and LOTOS relate, with interesting and promising results [**?**]. As could be seen in the example, translation from LOTOS to (Object-)Z also requires an interpretation of and refinement rules for specifications with internal operations, this is called weak refinement and is described in [**?**].

Tool support

It is important to be able to automate as much as possible of the unification and consistency checking process. Since the complexity and structure of these conditions is almost exclusively determined by the predicates that occur in the viewpoint specifications, existing methods for automated theorem proving in Z (e.g. [**?**, **?**, **?**]) can be used for proving the consistency conditions. With that in mind we have built a small prototype to support the process. A unification tool (described in [**?**]) was implemented using Generic Formaliser, a generalisation of a tool for Z written by Logica, called (Z Specific) Formaliser. In addition an implementation of Z in the theorem prover Isabelle has been used to provide theorem proving support for verification of consistency conditions. The consistency conditions can be automatically generated from the Z unification tool and fed into the Isabelle theorem prover.

Also it should be clear from the example in this paper that a significant part of the work involved is to do with *simplifying* predicates and specifications. This is a typical activity for tactic-based theorem proving systems: the user needs to have an idea of the sort of simplifications that might be possible, and then the system can aid in proving they are indeed correct.

9 Related work

Whereas our work on viewpoint consistency was motivated by the emerging standard reference model for open distributed processing (see section 1), the use of multiple viewpoints for specifying complex systems is not unique to the RM-ODP. Using different abstractions when reasoning about complex systems is an effective way of separating concerns. Not surprisingly, many other disciplines involved in information systems development have come up with similar approaches. View-integration in conceptual database design has been a widely researched topic in the 1980s [**?**, **?**]. The use of viewpoints in requirements engineering even dates back to the late 1970s [**?**] (an overview of viewpoint oriented approaches to requirements definition can be found elsewhere [**?**]). More recently, viewpoints have been proposed and researched for program development environments [**?**] and information systems design [**?**]. Within the software and requirements engineering communities there is currently a substantial number of researchers working on what is phrased as "the multiple perspectives problem" [**?**, **?**, **?**, **?**, **?**].

In any viewpoint oriented specification approach, the viewpoints will not be completely independent. Ultimately, each viewpoint is concerned with the same system, and constraints expressed in different specifications are likely to overlap. Therefore, viewpoint consistency has to be addressed by each realistic viewpoint oriented approach.

The viewpoint oriented methods mentioned above generally do not base their notion of consistency on development relations. Partly this is due to the fact that they use languages which are less formal or less development oriented than the ones we use. Consistency is often determined by explicit consistency relations on and between the viewpoints [**?**], based on overlap identification (akin to our correspondences) and similarity analysis. Unification, however, also seems a useful process for consistency checking in requirements engineering [**?**].

9.1 Formal methods for ODP viewpoints

9.1.1 Architectural semantics

The RM-ODP defines abstract languages for the five viewpoints. Several research groups have worked on populating this abstract framework with specific formal specification notations (e.g. [**?**, **?**, **?**]). In particular, work on the ODP *architectural semantics* aims to provide interpretations of the abstract modelling and specification concepts in a number of standardised formal description techniques [**?**, **?**, **?**].

The architectural semantics will provide the basis for uniform and consistent comparison between formal descriptions of the same system or standard in different FDTs. It is, therefore, of significance to achieve realistic consistency checking techniques.

On the other hand, inter-language consistency checking techniques, such as those developed in this paper, may also play a role in the definition of the architectural semantics. In order for the architectural semantics to act as a bridge between the ODP model and the semantic models of the FDTs, the architectural semantics should be consistent in two ways. Firstly, it is necessary to demonstrate that the interpretations of the same architectural entity in different FDTs are consistent. Secondly, the architectural semantics of different viewpoints are related and should therefore be checked for consistency. These are issues for further research.

9.1.2 Formal methods for consistency and unification

Some researchers circumvent the issues of consistency and unification by assuming an ordering between the ODP viewpoints. The information viewpoint is taken to be more abstract than the computational viewpoint. They subsequently define transformations from the former to the latter [**?**, **?**, **?**].

In our work on LOTOS we have mainly focused on the question of consistency and less on the unification problem. The composition of process specifications has been considered before by several others [**?**, **?**, **?**, **?**].

From amongst these approaches, perhaps the most common is to use the LOTOS parallel operator, $|[G]|$, as the notion of composition. This yields the so called constraint oriented style of specification [**?**] where the parallel composition of two specifications is viewed as its unification. However, such a notion of unification does not have nice formal properties. In particular, the main result on such constraint oriented composition is that the $||$ operator (i.e. $|[G]|$ with an empty gate set) respects trace preorder, but this is a very weak notion of development (see [**?**, **?**] for more details) and is not in general sufficient.

Ichikawa [**?**] introduced a 'specification merge operator', $\oplus$, to obtain a common extension (**ext**) of two processes. We define an operator in [**?**] which is an improvement of the $\oplus$-operator, in the sense that it can deal with non-deterministic specifications. Leduc [**?**] considers the balanced composition of processes with respect to conformance (**conf**) and reduction (**red**). The result is a denotational characterisation of the unification, which is close to the characterisation in terms of traces and refusals given in section **??**. Khendek [**?**] proposes an algorithm to compute a common extension (**ext**) of two processes modelled by acceptance graphs. Interestingly, this algorithm not only extends the original processes, but also preserves their cyclic traces. Consistency and unbalanced composition are, however, not considered by any of these authors.

In addition, it is clear from this previous work that there is a trade-off between the operational and denotational approaches. The operational approach provides a high-level composition operator, which has a similar status to the existing LOTOS operators. However, operational definitions

typically fail to characterise the least developed unification. In contrast, the denotational approach enables a natural characterisation of least development, but its status is quite different from the other LOTOS operators as it is interpreted in a different semantic setting.

A number of the problems with these approaches are resolved in [**?**] by separating refinement from non-determinism. In particular, in the resulting modal transition system based approach, an operational characterisation of the least unification can be given.

9.2 Viewpoints in Z and related methods

Various approaches to using Z as a language for partial specification have been described in the literature. For more detailed comparisons of the various approaches with our methods, we refer to [**?**, **?**].

Ainsworth, Wallis and others [**?**, **?**] advocate an approach very similar to ours. They use the term *amalgamation* for what we call unification, and *union* for operation unification, and relate these notions to a variant of refinement. However, they are less explicit about correspondences and consistency conditions.

In the more abstract relational framework of Milli, Frappier, et al [**?**, **?**] operation unification appears as well, under the name of *demonic join*. Their work also points out the link between viewpoint consistency and feature interaction.

D. Jackson [**?**] describes "view composition" in Z, giving many examples of syntactic constructions in Z that can be profitably used for partial specification, including promotion and a syntactic variant of unification. However, not all of these are relevant in a semantic sense.

Zave and Jackson describe in several papers [**?**, **?**] a multiparadigm specification technique, with impressive applications in specifications of telephone switching systems. Their work is similar to ours in that it uses Z and other languages for partial specification. For consistency checking, they use a translation of all specifications to first order predicate logic. Composition of partial specifications is then "just" conjunction [**?**].

Approaches in which Z specifications are augmented with specifications in other formalisms can also be viewed as specifications with multiple viewpoints, with consequences similar to those that follow from our work on comparing viewpoints in LOTOS and (Object-)Z. In particular, Fischer, Smith and Derrick combine Z with CSP [**?**, **?**, **?**], and Weber et al [**?**] combine Z with Statecharts. However, most methods that combine Z with some other language manage to avoid the consistency issue by the use of layering techniques, or by using the various languages in different stages of development. Kasurinen and Sere [**?**], for example, in their integration of Z and action systems use a layering technique, Z providing the types and operations to be used in the action systems descriptions.

References

[1] AFNOR. *A direct computational language semantics for Part 4 of the RM-ODP.* ISO/IEC JTC1/SC21/WG7 approved AFNOR contribution, July 1994.

[2] M. Ainsworth, A. H. Cruickshank, L. J. Groves, and P. J. L. Wallis. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, February 1994.

[3] M. Ainsworth, S.. Riddle, and P.J.L. Wallis. Formal validation of viewpoint specifications. *Software Engineering Journal*, 11(1):58–66, January 1996.

[4] D. Baldwin. Applying multiple views to information systems: A preliminary framework. *Data base*, 24(4):15–30, November 1993.

[5] C. Batini and M. Lenzerini. A methodology for data schema integration in the entity relationship model. *IEEE Transactions on Software Engineering*, SE-10:640–655, November 1984.

[6] C. Bernardeschi, J. Dustzadeh, A. Fantechi, E. Najm, A. Nimour, and F. Olsen. Transformations and consistent semantics for ODP viewpoints. In H. Bowman and J. Derrick, editors, *FMOODS'97, 2nd IFIP Conference on Formal Methods for Open Object Based Distributed Systems*. Chapman and Hall, July 1997.

[7] G.S. Blair and Jean-Bernard Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.

[8] E. Boiten. Z unification tools in Generic Formaliser. Technical Report 10-97, Computing Laboratory, University of Kent at Canterbury, 1997.

[9] E. Boiten, H. Bowman, J. Derrick, and M. Steen. Managing inconsistency and promoting consistency. In revision, available from `http://www.cs.ukc.ac.uk/research/tcs/consistency/tse.html`, September 1997.

[10] E.A. Boiten and J. Derrick. IO - refinement in Z. In A.S. Evans, D.J. Duke, and T. Clark, editors, *3rd BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer Verlag, September 1998.

[11] E.A. Boiten, J. Derrick, H. Bowman, and M. Steen. Coupling schemas: data refinement and view(point) composition. In D.J.Duke and A.S.Evans, editors, *2nd BCS-FACS Northern Formal Methods Workshop*, Workshops in Computing. Springer-Verlag, July 1997.

[12] E.A. Boiten, J. Derrick, H. Bowman, and M. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75, 1999.

[13] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.

[14] G. Booch. *Object-oriented Analysis and Design*. The Benjamin/ Cummings Publishing Company, Inc, 1994.

[15] N. Boudriga, F. Elloumi, and A. Mili. On the lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.

[16] J. P. Bowen and M. Gordon. Z and HOL. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop*, pages 141–167, Cambridge, July 1994. Springer-Verlag.

[17] H. Bowman, E. Boiten, J. Derrick, and M. Steen. Strategies for consistency checking based on unification. *Science of Computer Programming*, 33:261-298, April 1999.

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

[19] H. Bowman, J. Derrick, P. Linington, and M. Steen. Cross viewpoint consistency in Open Distributed Processing. *IEE Software Engineering Journal*, 11(1):44–57, January 1996.

[20] H. Bowman, J. Derrick, and M. Steen. Some results on cross viewpoint consistency checking. In K. Raymond and L. Armstrong, editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 399–412, Brisbane, Australia, February 1995. Chapman and Hall.

[21] H. Bowman, E.A.Boiten, J. Derrick, and M. Steen. Viewpoint consistency in ODP, a general interpretation. In E. Najm and J.-B. Stefani, editors, *First IFIP International workshop on Formal Methods for Open Object-based Distributed Systems*, pages 189–204, Paris, March 1996. Chapman & Hall.

[22] H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency. *Formal Methods in System Design*. To appear, 2000.

[23] E. Brinksma, G. Scollo, and C. Steenbergen. Process specification, their implementation and their tests. In B. Sarikaya and G. v. Bochmann, editors, *Protocol Specification, Testing and Verification, VI*, pages 349–360, Montreal, Canada, June 1986. North-Holland.

[24] TINA C. Telecommunications information networking architecture, 1997. WWW: http://www.tinac.com/.

[25] CCITT Z.100. *Specification and Description Language SDL*, 1988.

[26] E. Cusack. Object oriented modelling in Z for Open Distributed Systems. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 167–178, Berlin, Germany, September 1991. North-Holland.

[27] H.S. Delugach. An approach to conceptual feedback in multiple viewed software requirements modeling. In Finkelstein and Spanoudakis [**?**], pages 242–246.

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

[29] J. Derrick, E.A. Boiten, H. Bowman, and M.W.A. Steen. Viewpoints and Consistency - translating LOTOS to Object-Z. *Computer Standards and Interfaces*, 21:251–272, 1999.

[30] J. Derrick, H. Bowman, E. Boiten, and M. Steen. Comparing LOTOS and Z refinement relations. In *FORTE/PSTV'96*, pages 501–516, Kaiserslautern, Germany, October 1996. Chapman & Hall.

[31] J. Derrick, H. Bowman, and M. Steen. Viewpoints and Objects. In J. P. Bowen and M. G. Hinchey, editors, *Ninth Annual Z User Workshop*, LNCS 967, pages 449–468, Limerick, September 1995. Springer-Verlag.

[32] John Derrick, Eerke Boiten, Howard Bowman, and Maarten Steen. Specifying and Refining Internal Operations in Z. *Formal Aspects of Computing*, 10:125–159, December 1998.

[33] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, September 1995.

[34] J. Dustzadeh and E. Najm. Consistent semantics for ODP information and computational models. In T. Higashino and A. Togashi, editors, *FORTE/PSTV'97*, pages 107–126. Chapman & Hall, November 1997.

[35] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification*. Springer-Verlag, 1985.

[36] K. Farooqui and L. Logrippo. Viewpoint transformation. In J. de Meer, B. Mahr, and O. Spaniol, editors, *Open Distributed Processing II*, pages 352–362. IFIP TC6, September 1993.

[37] K. Farooqui and L. Logrippo. Viewpoint transformations. In J. de Meer, B. Mahr, and O. Spaniol, editors, *2nd International IFIP TC6 Conference on Open Distributed Processing*, pages 352–362, Berlin, Germany, September 1993.

[38] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal on Software Engineering and Knowledge Engineering, Special issue on Trends and Research Directions in Software Engineering Environments*, 2(1):31–58, March 1992.

[39] A. Finkelstein and G. Spanoudakis, editors. *SIGSOFT '96 International Workshop on Multiple Perspectives in Software Development (Viewpoints '96)*. 1996.

[40] A. Finkelstein, G. Spanoudakis, and D. Till. Managing interference. In Finkelstein and Spanoudakis [**?**], pages 172–174.

[41] A.C.W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.

[42] C. Fischer. 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.

[43] J. Fischer, A. Prinz, and A. Vogel. Different FDT's confronted with different ODP-viewpoints of the trader. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial Strength Formal Methods*, LNCS 670, pages 332–350. Springer-Verlag, 1993.

[44] M. Frappier, A. Mili, and J. Desharnais. Program construction by parts. In B. Möller, editor, *Mathematics of Program Construction: Third International Conference*, volume 947 of *Lecture Notes in Computer Science*, pages 257–281. Springer-Verlag, 1995.

[45] M.-C. Gaudel and J. Woodcock, editors. *FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1996.

[46] V. Gay, P. Leydekkers, and R. Huis in 't Veld. Specification of multiparty audio and video; interaction based on the reference model of open; distributed processing. *Computer Networks and ISDN Systems*, January 1995.

[47] R. Gotzhein and F. H. Vogt. The design of a temporal logic for Open Distributed Systems. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 229–240, Berlin, Germany, September 1991. North-Holland.

[48] J. J. Van Griethuysen. Enterprise modelling, A necessary basis for modern information systems. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 29–68, Berlin, Germany, September 1991. North-Holland.

[49] I. Hayes, M. Mowbray, and G.A. Rose. Signalling System No. 7 - The network layer. In E. Brinksma, G. Scollo, and C.A. Vissers, editors, *Protocol Specification Testing and Verification IX*, pages 3–14. North-Holland, 1989.

[50] H. Ichikawa, K. Yamanaka, and J. Kato. Incremental Specification in LOTOS. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification X*, pages 183–196, Ottawa, Canada, 1990.

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

[52] ISO 9074. *Estelle, a Formal Description Technique based on an extended state transition model*, June 1987.

[53] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing. ISO 10746, 1993. Part 1 to 4.

[54] ITU/ISO CD ISO 13235/ITU.TS Rec.9tr. *ODP Trading Function*, 1994.

[55] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, October 1995.

[56] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1989.

[57] V. Kasurinen and K. Sere. Integrating action systems and Z in a medical system specification. In Gaudel and Woodcock [**?**], pages 105–119.

[58] F. Khendek and G. von Bochmann. Merging behaviour specifications. *Journal of Formal Methods in System Design*, 6(3):259–294, June 1995.

[59] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher-Order Logics – 9th International Conference*, volume 1125 of *Lecture Notes in Computer Science*, pages 283–298, 1996.

[60] G. Kotonya and I. Sommerville. Viewpoints for requirements definition. *IEE Software Engineering Journal*, 7(6):375–387, November 1992.

[61] I. Kraan and P. Baumann. Implementing Z in Isabelle. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 7-9, 1995, Proceedings*, volume 967 of *LNCS*, pages 355–373. Springer-Verlag, 1995.

[62] G. Leduc. *On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS*. PhD thesis, University of Liège, Liège, Belgium, June 1991.

[63] P. F. Linington. RM-ODP The Architecture. In K. Raymond and L. Armstrong, editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 15–33, Brisbane, Australia, February 1995. Chapman and Hall.

[64] S. Meyers. Difficulties in integrating multiview development systems. *IEEE Software*, 8(1):49–57, January 1991.

[65] Microsoft. The Component Object Model specification, 1997. http://www.microsoft.com/oledev/olecom/title.htm.

[66] G. P. Mullery. CORE - a method for controlled requirement specification. In *4th International Conference on Software Engineering*, pages 126–135. IEEE Computer Society, 1979.

[67] E. Najm and J.-B. Stefani. Computational models for open distributed systems (invited talk). In H. Bowman and J. Derrick, editors, *2nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 157–176. Chapman & Hall, 1997.

[68] S. Navathe, R. Elmasri, and J. Larson. Integrating user views in database design. *IEEE Computer*, 19(1):50–62, January 1986.

[69] B. A. Nuseibeh. *A Multi-Perspective Framework for Method Integration*. PhD thesis, Imperial College, University of London, 1994.

[70] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1997. WWW: http://www.omg.org/.

[71] P. F. Pinto and P. F. Linington. A language for the specification of interactive and distributed multimedia applications. In B. Mahr J. de Meer and O. Spaniol, editors, *IFIP International Conference on Open Distributed Processing*, pages 217–234, Berlin, Germany, September 1993. North-Holland.

[72] W.L. Poon and A. Finkelstein. Consistency management for multiple perspective software development. In Finkelstein and Spanoudakis [**?**], pages 192–196.

[73] K. Raymond. Reference model of open distributed processing (RM-ODP): Introduction. In K. Raymond and L. Armstrong, editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 3–14, Brisbane, Australia, February 1995. Chapman and Hall.

[74] A. Reeves, M. Marashi, and D. Budgen. A software design framework or how to support real designers. *IEE Software Engineering Journal*, 10(4):141–155, July 1995.

[75] J. Ronayne. *The Integrated Services Digital Network: from concept to application*. Pitman, London, 1987.

[76] M. Van Sinderen and J. Schot. An engineering approach to ODP system design. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 301–312, Berlin, Germany, September 1991. North-Holland.

[77] R. Sinnott. *An Initial Architectural Semantics in Z of the Information Viewpoint Language of Part 3 of the ODP-RM*. ISO/IEC SC21/WG7 N915, July 1994. BSI Input document to the ODP Plenary meeting in Southampton.

[78] R.O. Sinnott and K.J. Turner. Applying formal methods to standard development: The open distributed processing experience. *Computer Standards and Interfaces*, 17:615–630, 1995.

[79] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[80] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME '97)*, LNCS 1313, pages 62–81, Graz, Austria, September 1997. Springer-Verlag.

[81] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, pages 293–302, Hiroshima, Japan, November 1997. IEEE Computer Society.

[82] I. Sommerville. *Software Engineering*. Addison-Wesley, 1989.

[83] J. M. Spivey. *The Z notation: A reference manual*. Prentice Hall, 1989.

[84] M. W. A. Steen, H. Bowman, and J. Derrick. Composition of LOTOS specifications. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification, XV*, pages 73–88, Warsaw, Poland, 1995. Chapman & Hall.

[85] M. W. A. Steen and J. Derrick. Formalising ODP Enterprise Policies. In *3rd International Enterprise Distributed Object Computing Conference (EDOC '99)*, University of Mannheim, Germany, September 1999. IEEE Publishing.

[86] M.W.A. Steen. *Consistency and Composition of Process Specifications*. PhD thesis, University of Kent at Canterbury, United Kingdom, 1998.

[87] M.W.A. Steen and J. Derrick. Applying the UML to the ODP enterprise viewpoint. Technical Report 8-99, Computing Laboratory, University of Kent at Canterbury, May 1999.

[88] S. Stepney, D. Cooper, and J. Woodcock. More powerful Z data refinement. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 284–307. Springer-Verlag, September 1998.

[89] C.N. Taylor, M.W.A. Steen, J. Derrick, and E.A. Boiten. Library case study. In preparation, 2000.

[90] K. Turner, (Ed.), *Computer networks and ISDN Systems*, 1995. Special Issue.

[91] R.J. van Glabbeek. The refinement theorem for ST-bisimulation semantics. In *Programming Concepts and Methods*. Elsevier Science Publishers, 1990.

[92] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. On the use of specification styles in the design of distributed systems. *Theoretical Computer Science*, 89(1):179–206, October 1991.

[93] M. Weber. Combining statecharts and Z for the design of safety-critical control systems. In Gaudel and Woodcock [**?**], pages 307–326.

[94] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

[95] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[96] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.

[97] P. Zave and M. Jackson. Where do operations come from? A multiparadigm specification technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, July 1996.

[98] H. Zimmermann. OSI - reference model - ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28:425–432, 1980.