Viewpoint consistency in Z and LOTOS: A case study

Eerke Boiten, Howard Bowman, John Derrick and Maarten Steen

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Email: E.A.Boiten@ukc.ac.uk.)

Abstract. Specification by viewpoints is advocated as a suitable method of specifying complex systems. Each viewpoint describes the envisaged system from a particular perspective, using concepts and specification languages best suited for that perspective.
Inherent in any viewpoint approach is the need to check or manage the *consistency* of viewpoints and to show that the different viewpoints do not impose contradictory requirements. In previous work we have described a range of techniques for consistency checking, refinement, and translation between viewpoint specifications, in particular for the languages LOTOS and Z. These two languages are advocated in a particular viewpoint model, viz. that of the Open Distributed Processing (ODP) reference model. In this paper we present a case study which demonstrates how all these techniques can be combined in order to show consistency between a viewpoint specified in LOTOS and one specified in Z.

Keywords: Viewpoints; Consistency; Z; LOTOS; ODP.

1 Introduction

Specification by viewpoints is advocated as a structuring method for the description of large software systems [14]. One advantage of this method of specification is a true separation of concerns, due to each viewpoint representing only one perspective on the envisaged system. Additionally, each viewpoint can use a specification language which is dedicated to its particular perspective – acknowledging the generally held belief that no formal method applies well to all problem domains.

Our motivation for studying viewpoint specification derives from its use in distributed system design, in particular in the Open Distributed Processing (ODP) standard [18]. There are five viewpoints with fixed pre-determined roles in ODP: *enterprise*, *information*, *computational*, *engineering* and *technology*. The perspectives they represent are at potentially different levels of abstraction (this is in contrast to many other viewpoint models). For example, the *computational* viewpoint is concerned with the algorithms and data flow of the distributed system function. It represents the system and its environment in terms of objects which interact by transfer of information via interfaces. The *engineering* viewpoint, on the other hand, is more concerned with distribution mechanisms, and defines the building blocks which can be combined to provide the system's functionality.

Inherent in any viewpoint approach like ODP's is the need to check or manage the *consistency* of viewpoints and to show that the different viewpoints do not impose contradictory requirements. The mechanisms needed to do this depend on the viewpoint languages used. Consistency checking becomes particularly challenging when the viewpoints are described in different specification languages or even according to different paradigms. Of the available formal techniques we are interested in the use of Z and LOTOS, due to their potential use in specific ODP viewpoints and also because they are representative of different kinds of specification languages.

In previous papers we have described a number of individual techniques and aspects of consistency checking: a general framework for defining consistency [6], techniques for LOTOS [24], techniques for Z [3, 10], and techniques for relating LOTOS and Z [13, 11]. However, so far these have not been brought together in a single case study. In this paper we present such a case study: existing techniques will be combined in an example, demonstrating how consistency can be shown between one viewpoint specified in LOTOS, and another specified in Z.

This paper (sections 3-6) illustrates each of these techniques with reference to our running example of a protocol specification (introduced in section 2). By combining these techniques we check the consistency of an engineering viewpoint written in Z with a computational viewpoint written in LOTOS as follows. We first translate the LOTOS specification to an observationally equivalent one in Z, then we check the consistency of the two viewpoints now both expressed in Z. The constructive method used for this results in a common refinement of the two Z viewpoints, whose existence demonstrates consistency of the original viewpoints.

However, these mechanisms largely deal with viewpoints written at the same level of abstraction, and they need to be extended to deal with the differing levels of abstraction found in various viewpoints. The final section of the paper discusses what support might be made available by using appropriate specification styles or methods of refinement that are compatible with viewpoint modelling and consistency checking.

2 A simple example

We illustrate our work by reference to a simple example, which we outline in this section. The example we describe specifies a communications protocol from two ODP viewpoints - a computational viewpoint and an engineering viewpoint (although the fit is not perfect). The example is based on the specification of the Signalling System No. 7 protocol described in [28]. Because the engineering viewpoint in this example is heavily state dependent we have specified it in Z. However, the choice of language in the viewpoints is immaterial to the essence of the work described here.

2.1 The Computational Viewpoint in LOTOS

Suppose the protocol handles messages of type *element*, which contains a distinguished value *null*. The protocol is described here in terms of two sequences *in* and *out* (which represent messages that have arrived in the protocol (*in*), and those that have been forwarded (*out*)). Incoming messages are added to the left of *in*, and the messages contained in *in* but not in *out* represent those currently inside the protocol. The specification ensures that the *out* sequence is a suffix of the *in* sequence, so that the protocol delivers without corrupting or re-ordering.

The data typing part of the LOTOS specification defines the sort *seq* to represent sequences and its associated operations algebraically in the usual fashion. The equations defining the operations on sequences have been omitted, since most of them are standard. A less traditional one is suffix subtraction: $x - y = z$ iff x is the concatenation of z and y.

Two actions model the behaviour of the protocol, which describe the transmission and reception of messages. *transmit* accepts a new message and adds it to the *in* sequence. The *receive* action either delivers the latest value as an output (which is then also added to the output sequence), or a null value is output, modelling the environment's "busy waiting" (in which case *out* is unaltered). Initially, no messages have been sent. This viewpoint is specified as follows.

specification
type seq **is** element, bool **with**
  sorts *seq*
  opns $empty\_seq :\to seq$
    $add : element, seq \to seq$
    $\neq: seq, seq \to bool$
    $first : seq \to element$
    $last : seq \to element$
    $front : seq \to seq$
    $- : seq, seq \to seq$
    $cat : seq, seq \to seq$
    $\# : seq \to nat$
  eqns
    (* definition of operations omitted *)
endtype
behaviour
  $Protocol[transmit, receive](empty\_seq, empty\_seq)$
where
  process $Protocol[transmit, receive](in, out : seq)$: **noexit** :=
  $transmit?x : element;\ Protocol[transmit, receive](add(x, in), out)$
  []
  $receive!null;\ Protocol[transmit, receive](in, out)$
  []
  $[in \neq out] \to receive!last(in - out);$
    $Protocol[transmit, receive](in, add(last(in - out), out))$
  endproc
endspec

An alternative, but equally acceptable, specification at this level of abstraction would be to require that *receive* has some (non-*null*) effect as long as there are still messages within the system. To model this we would add a guard $[in = out]$ to the second branch of the choice. This specification, in fact, is in itself composed of two LOTOS specifications of parts of its behaviour, cf. section 6. We will see the consequences for consistency checking of this seemingly small change later.

2.2 The Engineering Viewpoint in Z

This engineering viewpoint describes the route the messages take through the medium in terms of a number of sections represented by a non-empty sequence of signalling point codes (SPC). Each section may send and receive messages of type M, and those that have been received but not yet sent on are said to be in the section. The messages pass through the sections in order. In the state schema, $ins\ i$ represents the messages currently inside section i, $rec\ i$ the messages that have been received by section i, and $sent\ i$ the messages that have been sent onwards from section i. The state and initialization schemas are then given by

$$[M, SPC]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Section}} \\
\hline
route : \mathrm{iseq}\ SPC \\
rec, ins, sent : \mathrm{seq}(\mathrm{seq}\ M) \\
\hline
route \neq \langle\,\rangle \\
\#route = \#rec = \#ins = \#sent \\
rec = ins \frown\frown sent \\
front\ sent = tail\ rec \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitSection}} \\
\hline
Section' \\
\hline
\forall\, i : \mathrm{dom}\ route \bullet \\
\quad rec\ i = ins\ i = sent\ i = \langle\,\rangle \\
\hline
\end{array}$$

where $\frown\frown$ denotes pairwise concatenation of the two sequences (so for every i we have $rec\ i = ins\ i \frown sent\ i$). The predicate $front\ sent = tail\ rec$ ensures that messages that are sent from one section are those that have been received by the next. This specification also has operations to transmit and receive messages, and they are specified as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transmit}} \\
\hline
\Delta Section \\
m? : M \\
\hline
route' = route \\
head\ rec' = \langle m? \rangle \frown (head\ rec) \\
tail\ rec' = tail\ rec \\
sent' = sent \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Receive}} \\
\hline
\Delta Section \\
m! : M \\
\hline
route' = route \wedge rec' = rec \\
front\ ins' = front\ ins \\
last\ ins' = front(last\ ins) \\
front\ sent' = front\ sent \\
m! = last(last\ ins) \\
last\ sent' = \langle m! \rangle \frown (last\ sent) \\
\hline
\end{array}$$

In this viewpoint, the new message received is added to the first section in the route in *Transmit*, and *Receive* will deliver from the last section in the route. In the computational viewpoint, messages arrive non-deterministically, but in this viewpoint the progress of the messages through the sections is modelled explicitly. To do this we use an internal action *Daemon* which chooses which section will make progress in terms of message transmission. The oldest message is then transfered to the following section, and nothing else changes. The important part of this operation is:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Daemon}} \\
\Delta \mathit{Section} \\
\hline
\exists\, i : 1..\#\mathit{route} - 1 \mid \mathit{ins}\ i \neq \langle\,\rangle \bullet \\
\qquad \mathit{ins}'\, i = \mathit{front}(\mathit{ins}\ \ i) \\
\qquad \mathit{ins}'(i+1) = \langle \mathit{last}(\mathit{ins}\ \ i)\rangle \frown \mathit{ins}(i+1) \\
\qquad \forall\, j : \mathrm{dom}\, \mathit{route} \mid j \neq i \land j \neq i+1 \bullet \mathit{ins}'j = \mathit{ins}\ j \\
\hline
\end{array}
$$

3 Consistency and Correspondences

In order to be able to check the consistency of multiple viewpoint specifications (such as the two just presented) 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 [6]. 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 (e.g. the viewpoints above both specify the result of receiving a message) 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.

What are the correspondences in the above example? Certainly the protocol transmits one type of message, so M and *element* should be identified. 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 inputs and outputs of the operations are identified). Finally, it is clear that *in* and *out* in the computational viewpoint in some way represent information that is also represented by *rec*, *ins* and *sent* in the engineering viewpoint. However, unlike the other correspondences this is not a matter of simply identifying these components. We note that they are related via the following predicate: $\mathit{head\ rec} = \mathit{in} \land\ \mathit{last\ sent} = \mathit{out}$. These correspondences can then be documented succinctly as a relation

$$
\{(M, \mathit{element}), (\mathit{Transmit}, \mathit{transmit}), (\mathit{Receive}, \mathit{receive}), (\mathit{head}\ \ \mathit{rec}, \mathit{in}),\\
(\mathit{last}\ \ \mathit{sent}, \mathit{out})\}
$$

Consistency. The concept of a *development relation* plays a key role in our definition of consistency. Such relations relate specifications during the development process. Many different development relations occur in practice, each with different fundamental properties, e.g. *conformance relations*, *refinement relations*, *equivalence relations* and *translations*. The latter of these enables different languages to be moved between, by translating from the syntax of one to the syntax of the other in such a way that the semantics are preserved.

Using the concept of a development relation, we can define consistency:

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.

Least Developed Unification. Besides a definition of consistency, we have also investigated methods for constructively establishing consistency [4]. 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 that all other unifications are developments of. 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. [4] considers the properties that development relations must possess for such unifications to exist. In most cases development relations possess the required properties (in particular, Z refinement produces a least developed unification) and as a reflection of this, we will use a least developed unification strategy below in order to check the consistency of the protocol viewpoints.

4 Relating LOTOS and Z

Comparing viewpoints written in LOTOS and 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 so far is to adopt a more behavioural interpretation of Z. We do so by using an object-oriented variant of Z called ZEST [8], developed by British Telecom specifically to support distributed system specification. ZEST does not increase the expressive power of Z, and a flattening to Z is provided. This enables us to produce output in a standardised language, whilst supporting the need to provide object-based capabilities in formal techniques used within ODP.

Object-based languages have a natural behavioural interpretation, and we have exploited this by defining a common semantics for LOTOS and a subset of Z in an extended transition system, which is used to validate a *translation* from full LOTOS into Z [13]. The essential idea behind the translation is to turn LOTOS processes into ZEST objects, and hence if necessary into Z.

The definition of *element* (which was omitted in the LOTOS specification) would be translated to a definition of *element* in Z, for example:

$[element]$

$$\begin{array}{|l} null : element \end{array}$$

For the data typing part, the ADT component of a LOTOS specification is translated directly into the Z type system. For example, the above LOTOS viewpoint's ADT can be translated directly to an axiomatic declaration in Z, viz:

$[seq]$

$$\begin{array}{|l}
empty_seq : seq \\
add : element \times seq \rightarrow seq \\
last : seq \rightarrow element \\
\hline
\forall x, y : element, q : seq \bullet\ last(add(x, empty_seq)) = x \\
\qquad\qquad\qquad\qquad \wedge\ last(add(x, add(y, q))) = last(add(y, q))
\end{array}$$

Moreover, any realistic consistency checking toolbox will also contain direct translations from axiomatic descriptions of standard structured types (e.g. sets and sequences) into their Z mathematical toolbox (cf. [23]) equivalents. We will assume that this translation has indeed been made in this example (and hence identify *empty_seq* and $\langle\rangle$).

For the LOTOS behaviour expression, we first derive its representation in the common semantic model (the details of the algorithm need not concern us here), and use this to generate the Z specification. This will involve translating each LOTOS action into a ZEST operation schema with explicit pre- and post-conditions to preserve the temporal ordering. Note that we assume (as usual in ZEST) a firing condition interpretation [25] of operation pre-conditions to ensure the interpretation of LOTOS actions corresponds correctly to that of Z operations.

For example, the above LOTOS viewpoint will be translated into a Z specification which contains operation schemas with names *transmit* and *receive*. The operation schemas have appropriate inputs and outputs (controlled by channels *ch*? and *ch*!) to perform the value passing defined in the LOTOS process. Each operation schema includes a predicate (defined over the state variable s) to ensure that it is applicable in accordance with the temporal behaviour of the LOTOS specification. Thus the behaviour expression in the above viewpoint is translated to the following Z schemas.

$$\begin{array}{|l}
\multicolumn{1}{l}{State} \\
\hline
s : \{s_0, s_1, s_2, s_3\} \\
in, out : seq \\
x : element \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Init_State} \\
\hline
\Delta State \\
\hline
s' = s_0 \\
in' = empty_seq \\
out' = empty_seq \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{transmit}} \\ \hline
\Delta State \\
ch? : element \\ \hline
s = s_0 \land s' = s_1 \land x' = ch? \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{receive}} \\ \hline
\Delta State \\
ch! : element \\ \hline
(s = s_0 \land s' = s_2 \land ch! = null) \lor \\
(in \neq out \land s = s_0 \land s' = s_3 \\
\ \land ch! = last(in - out)) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{i} \\ \hline
\Delta State \\ \hline
(s = s_1 \land s' = s_0 \land in' = add(x, in) \land out' = out) \lor \\
(s = s_2 \land s' = s_0 \land in' = in \land out' = out) \lor \\
(s = s_3 \land s' = s_0 \land in' = in \land out' = add(last(in - out), out)) \\ \hline
\end{array}$$

Because the translation was defined indirectly via the semantics, recursion is dealt with by using an internal action, which is translated as an internal Z operation with special name i. However, we can re-write it without the internal action by replacing the three operation schemas by the following two. In order to reason about Z specifications which contain internal actions we have defined a generalisation of refinement in Z called weak Z-refinement [10], and the specification without the above internal operation is weak Z-refinement equivalent to the original.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{transmit}} \\ \hline
\Delta State \\
ch? : element \\ \hline
in' = add(ch?, in) \land out' = out \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{receive}} \\ \hline
\Delta State \\
ch! : element \\ \hline
in' = in \\
(out' = out \land ch! = null) \lor \\
(in \neq out \land ch! = last(in - out) \land \\
\ out' = add(ch!, out)) \\ \hline
\end{array}$$

The two viewpoints are now both expressed in Z, and the following section shows how we can check them for consistency. However, knowing that both viewpoints are consistent (after translation) with respect to Z refinement may not always be enough. The LOTOS viewpoint had an associated development relation, which does not necessarily correspond to Z refinement under translation. Thus, we have begun to investigate how the development relations in Z and LOTOS relate, with interesting and promising results [11]. For example, a failure-traces reduction in a LOTOS viewpoint will imply a Z-refinement after translation into Z.

5 Consistency in Z

Now the two viewpoints are specified in Z, we can apply the consistency checking techniques for Z described in [3]. This involves constructing a least refined unification of the two viewpoints, in two phases. In the first phase ("state unification"), a unified state space (i.e., a state schema) for the two viewpoints has to be constructed. The essential components of this unified state space are the correspondences between the types in the viewpoint state spaces. The viewpoint operations are then adapted to operate on this unified state. At this stage we have to check that a condition called *state consistency* is satisfied. In the second phase, called *operation unification*, pairs of adapted operations from the viewpoints which are linked by a correspondence (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.

5.1 State unification

To simplify the presentation, we replace the state space of the computational viewpoint by the following[1], which is a reversible data-refinement step that excludes some unreachable states (note that *out* being a suffix of *in* is indeed an invariant of the computational viewpoint). It also removes the component x which has become superfluous once the internal operation has been removed.

$$\begin{array}{|l}
\hline
\textit{NState} \\
\hline
\textit{in}, \textit{out} : \mathrm{seq}\, M \\
\hline
\textit{out} \in \mathrm{suffixes}\ \textit{in} \\
\hline
\end{array}$$

We describe the correspondences between the two viewpoints (see section 3) as a schema between the state spaces *Section* and *NState*, this is then used to build the unified state schema.

$$R \mathrel{\widehat=} [\,\textit{Section}; \textit{NState} \mid \textit{head}\ \textit{rec} = \textit{in} \land \textit{last}\ \textit{sent} = \textit{out}\,]$$

R is total in both directions, we prove this by showing that it includes a total function in both directions.

- From *Section* to *NState* : $R = \lambda\, \textit{rec}, \textit{ins}, \textit{sent}, \textit{route} \bullet (\textit{head}\ \textit{rec}, \textit{last}\ \textit{sent})$. This is a total function since *Section* ensures that *rec* and *sent* are non-empty, and also that *last sent* is a suffix of *head rec*.

[1] A formal definition in Z of suffixes would be

$$\begin{array}{|l}
\hline
[X] \\
\hline
\mathrm{suffixes} : \mathrm{seq}\, X \rightarrow \mathbb{P}\,\mathrm{seq}\, X \\
\hline
\forall\, x, y : \mathrm{seq}\, X \bullet y \in \mathrm{suffixes}\, x \Leftrightarrow \textit{rev}\ y \subseteq \textit{rev}\ x \\
\hline
\end{array}$$

which makes use of sequences being particular sets, on which set inclusion turns out to be the prefix relation.

– From *NState* to *Section*: $R \supseteq \lambda\, in, out \bullet (rr, ii, ss, rt)$, for example choosing all sections empty except for the first one, i.e. $head\ ii = in - out$; $head\ rr = in$; $\frown/(tail\ ii) = \langle\rangle$; $\mathrm{ran}\, ss = \mathrm{ran}(tail\ rr) = \{out\}$.

[3] describes how a correspondence relation needs to be totalised in order to form a correct unified state space; however, as the correspondence relation here is total in both directions it can be used directly to form the unified state. The condition of state consistency, viz. that the viewpoint state predicates are equivalent for any pair of states in R, is guaranteed to hold in this example because R's predicate includes the viewpoint predicates. Thus, the unified state space of the protocol viewpoints is

$$
\begin{array}{|l}
\hline
\textit{Un} \\
\textit{Section} \\
\textit{NState} \\
\hline
head\ rec = in \\
last\ sent = out \\
\hline
\end{array}
$$

which is essentially *Section* extended with derived components *in* and *out*.

The totality of R also greatly simplifies operation adaptation. All the viewpoint operations are adapted simply by making them operate on the unified state. The engineering viewpoint operations will thus become

$AdTransmit \mathrel{\widehat=} [\,\Delta Un \mid Transmit\,]$
$AdReceive \mathrel{\widehat=} [\,\Delta Un \mid Receive\,]$
$AdDaemon \mathrel{\widehat=} [\,\Delta Un \mid Daemon\,]$

The *AdDaemon* operation plays no further role: it is not linked to any operation from the computational viewpoint by a correspondence, so its adaption is already part of the unified specification and automatically consistent. The external operations will become, similarly

$Adtransmit \mathrel{\widehat=} [\,\Delta Un \mid transmit\,]$
$Adreceive \mathrel{\widehat=} [\,\Delta Un \mid receive\,]$

Now we have adapted the operations, we apply operation unification to the receive and transmit operations.

5.2 Operation unification and consistency

The general rule for operation unification is as follows [1, 3]. Two operations $Op1$ and $Op2$, both changing state S and with input $x? : T$, are unified to

$$
\begin{array}{|l}
\hline
\textit{Op} \\
\Delta S \\
x? : T \\
\hline
\mathrm{pre}\, Op1 \Rightarrow Op1 \\
\mathrm{pre}\, Op2 \Rightarrow Op2 \\
\hline
\end{array}
$$

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.

For the *transmit* operations, both adapted operations are total, i.e. their pre-conditions always hold. The unified transmit operation will thus be

$$unTransmit \mathrel{\widehat=} [\, \Delta Un \mid AdTransmit \wedge Adtransmit \,]$$

It remains to check that $unTransmit$ is satisfiable whenever both pre-conditions hold. For this, it is only necessary to check that the "derived" components in' and out' get consistent values: their new values can be computed from the old values of rec and $sent$ via Un' and $AdTransmit$, or via $Adtransmit$ and Un. These turn out to give the same values:

$$\begin{array}{ll}
in' & out' \\
\quad = \{ Un' \} & \quad = \{ Un \} \\
head\ rec' & last\ sent' \\
\quad = \{ AdTransmit \} & \quad = \{ AdTransmit \} \\
m? \frown head\ rec & last\ sent \\
\quad = \{ Un \} & \quad = \{ Un \} \\
m? \frown in & out \\
\quad = \{ Adtransmit \} & \quad = \{ Adtransmit \} \\
in' & out'
\end{array}$$

Thus, the unified transmit operation refines both original operations, and therefore the original *transmit* operations were consistent.

For the *receive* operations, consistency checking is a little more complicated. The adapted *receive* operation is total, however, *AdReceive* is only defined when there is a message in the last section. Thus, the unified receive operation is

$$\begin{array}{|l}
\multicolumn{1}{l}{unReceive} \\ \hline
\Delta Un \\
m! : M \\ \hline
last\ ins \neq \langle\,\rangle \Rightarrow Receive \\
in' = in \\
(out' = out \wedge m! = null) \vee \\
(in \neq out \wedge m! = last(in - out) \wedge out' = \langle m! \rangle \frown out) \\ \hline
\end{array}$$

Consistency is again determined by *Adreceive* and *AdReceive* both being satisfiable when both pre-conditions (so in this case, $last\ ins \neq \langle\,\rangle$) hold. What needs to be checked is the values of the derived components in' and out', like above for the transmit operations. The value for $m!$ in that case would also need to be checked, but since it is in both cases identical to the value which is added to *out* this causes no extra complications. For in', it is simply

$$in' = head\ rec' = head\ rec = in = in'$$

(according to Un', $AdReceive$, Un, and $Adreceive$), whereas for out' we have

$$\begin{array}{l} out' \\ \qquad = \{Un\} \\ last\ sent' \\ \qquad = \{AdReceive\} \\ last(last\ ins) \frown last\ sent \\ \qquad = \{Un\} \\ last(last\ ins) \frown out \\ \qquad = \{Section, Un\} \\ last((last\ rec) - out) \frown out \end{array}$$

$$\begin{array}{l} out' \\ \qquad = \{Adreceive\} \\ last(in - out) \frown out \end{array}$$

It can indeed be proved that, whenever both $(last\ rec) - out$ and $in - out$ are defined, their last elements are equal – this is essentially the proof that the sectional viewpoint does not distort the order in which elements travel through the protocol.

In this case, the specifications turn out to be consistent. However, with two minor but reasonable modifications they are not. Consider the alternative computational viewpoint mentioned in section 2.1; its *receive* operation would be translated from LOTOS to the following impatient receive in Z:

$$\begin{array}{|l} \hline \textit{impreceive} \\ \Delta NState \\ m! : element \\ \hline in' = in \\ (in = out \land out' = out \land m! = null) \lor \\ (in \neq out \land m! = last(in - out) \land out' = \langle m! \rangle \frown out) \\ \hline \end{array}$$

If we also modify the engineering viewpoint's receive operation to be total, by making it have no effect outside its precondition except for returning a *null*, i.e.

$$TotReceive \mathrel{\widehat{=}} Receive \lor (\neg\ \mathrm{pre}\ Receive \land \Xi Section \land m! = null)$$

the resulting specifications become *inconsistent*. When the last section is empty, but there is a message in some other section, $TotReceive$ will insist that the state remain unchanged. However, in that situation *impreceive* states that this message should be added to *out*. Unsurprisingly, the only way to prevent this situation and make these operations consistent is to ensure there is no more than one section... clearly not what was intended by the viewpoint specifiers.

6 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 [5, 24, 6], which is summarised in figure 1. The development relations highlighted are the following:

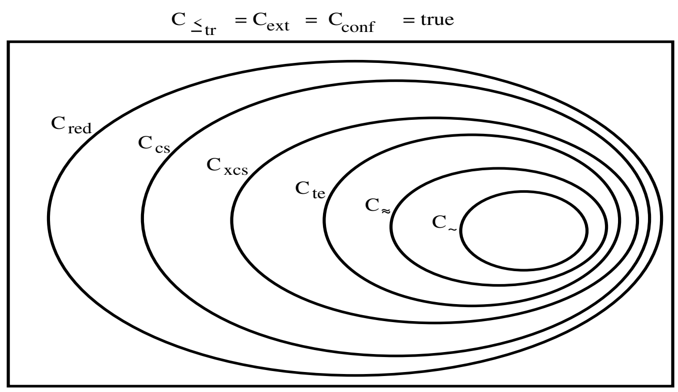

Fig. 1. LOTOS Consistency Relations

$\leq_{tr}$ *- 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),* $cs = conf \cap conf^{-1}$, $xcs = ext \cap conf^{-1}$, *te - testing equivalence,* $\approx$ *- weak bisimulation equivalence,* $\sim$ *- 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 undiscriminating, 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. Due to space limitations the illustration is slightly artificial, but it will serve to highlight our approach.

The LOTOS process *Protocol* presented in section 2 can be viewed as a unification of two partial specifications. The first is a generic protocol, called *GenProt*:

process $GenProt[send, deliver](queue : seq)$: **noexit** :=
$send?x : element;\ GenProt[send, deliver](add(x, queue))$
[]
$[queue \neq empty\_seq] \rightarrow deliver!last(queue);$
$GenProt[send, deliver](front(queue))$
endproc

which has a very simple behaviour, that of a queue which can grow arbitrarily; it is well known that the observable behaviour of a protocol can be viewed as a queue. *GenProt* offers two actions *send* and *deliver*. The first allows a user to input an element into the protocol, while the second enables a user to output an element from the other end of the protocol. As expected, a *deliver* action can only occur if the *queue* is not empty.

We envisage that such a description might be made available in a library of generic specification components. In addition, the ADT definition presented in section 2 with *Protocol* is assumed to be generic and hence available to all the partial specifications we consider here.

We also assume the following partial specification:

process $BusWait[transmit, receive](q : seq)$: **noexit** :=
$receive!null;\ BusWait[transmit, receive](q)$
endproc

which models the busy waiting behaviour that we see in *Protocol*. Specifically, *BusWait* will continually offer the action *receive!null*, denoting the possibility that the user attempts to output an element from the protocol when it is not ready to do so.

Assuming the following correspondences between the partial specifications,

$$\{(send, transmit), (deliver, receive), (queue, q)\}$$

we can view the second specification, *BusWait*, as defining additional behaviour which can be used to specialize the generic protocol *GenProt*. Implicitly, in this example, the behaviour of each partial specification extends the behaviour of the other specification. For example, *BusWait* adds the ability to perform the action *deliver*!*null* to the behaviour of *GenProt*. Thus, the appropriate refinement relation to use when unifying these two partial specifications is extension, *ext*.

Research based on work performed by Leduc [21] can be used to characterise the least developed unification according to extension. 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 [7]):

$$
\begin{array}{l}
Tr(U) = Tr(P) \cup Tr(Q) \quad \wedge \\
\quad \forall \sigma \in Tr(U), \\
\qquad \sigma \in Tr(P) \cap Tr(Q) \Longrightarrow Ref(U,\sigma) = Ref(P,\sigma) \cap Ref(Q,\sigma) \ \wedge \\
\qquad \sigma \in Tr(P) - Tr(Q) \Longrightarrow Ref(U,\sigma) = Ref(P,\sigma) \ \wedge \\
\qquad \sigma \in Tr(Q) - Tr(P) \Longrightarrow Ref(U,\sigma) = Ref(Q,\sigma)
\end{array}
$$

In fact, it can be shown that up to testing equivalence the following process:

```
process Prot[transmit, receive](q : seq) : noexit :=
  transmit?x : element; Prot[transmit, receive](add(x, q))
  []
  [q = empty_seq] → receive!null; Prot[transmit, receive](q)
  []
  [q ≠ empty_seq] → receive!last(q); Prot[transmit, receive](front(q))
endproc
```

is the least developed unification of *GenProt* and *BusWait*. Notice that this unification has inherited the naming of *BusWait*, the "specializing" process. Thus, we have assumed that our correspondences define a renaming of *GenProt*.

Finally, we can observe that *Prot* has the same trace/refusal characterisation as the process:

```
process Protocol[transmit, receive](in, out : seq) : noexit :=
  transmit?x : element; Protocol[transmit, receive](add(x, in), out)
  []
  [in = out] → receive!null; Protocol[transmit, receive](in, out)
  []
  [in ≠ out] →receive!last(in − out);
                Protocol[transmit, receive](in, add(last(in − out), out))
  endproc
```

which is the alternative LOTOS specification that we considered in section 2. The reason that *Protocol* and *Prot* are equivalent is that their externally visible behaviour is the same. This can be observed from the fact that at all corresponding points in the behaviour of the two processes, $in - out$ equals q.

7 Issues in viewpoint specification and refinement

The key component of the consistency checking strategy presented here is to be able to identify common refinements of the multiple 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 *out* 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.

Such changes of abstraction level add extra complications to viewpoints modelling and consistency checking, since the models of the two viewpoints are expressed in terms of different (but non-independent) primitives, thus, hindering the search for a common model. Potential solutions include the use of appropriate styles of specification and suitable methods of refinement.

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 computational viewpoint written in LOTOS one branch dealt with reception of a message[2]:

$$receive;\ Protocol[transmit, receive](in, add(last(in - out), out))$$

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

$$reclyr_1;\ \ldots;\ reclyr_n;\ Protocol[transmit, receive](in, add(last(in - out), out))$$

where the action *receive* has been action refined into the "partial behaviour" $reclyr_1;\ \ldots;\ reclyr_n$.

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.

[2] For simplicity, we have removed *receive*'s output here.

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. Action refinement has proved a hard problem to resolve. 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, the assumption of atomic actions is lost.

Current research has suggested that true concurrency models are better behaved in the presence of action refinement [26], since true concurrency models do not rely on the assumption of atomic actions. We are currently investigating the feasibility of moving to a true concurrency setting in order to integrate action refinement into consistency checking.

Promotion is a specification style often used in Z specifications for combining specifications at different levels of abstraction. Promotion works when a global operation on a number of components is defined in terms of a local operation on a single component. Using a special promotion schema *Promote*, the global operation is then defined by (where *Local* is the local state etc): $\exists \Delta Local \bullet Promote \land LocalOperation$. Promotion can also be used to define global operations in terms of multiple (possibly different) local operations on different instances of the local state, i.e. they may change the state of several local components at once.

This technique can be profitably used for specifying *viewpoints* at different levels of abstraction, in a manner which automatically guarantees their consistency: one viewpoint defines the global operations, but also this viewpoint includes the local state and its operations – but only their signatures (in Z terms, by including them as empty schemas). The global viewpoint thus does not make any assumptions about the local state and operations apart from their existence. Another viewpoint will then actually *define* the local state and its operations, see [12, 19] for examples of this in practice.

For example, suppose an additional viewpoint is to be combined with the engineering viewpoint of the protocol. This viewpoint wishes to use a number of protocols on a collection of routes (identified by *Id*), with an operation, *GlobalTransmit*, to transmit a message down a chosen route. We can specify the viewpoint as follows:

$$\begin{array}{|l} \multicolumn{1}{l}{Section} \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{Transmit} \\ \hline \end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Global} \\
series : Id \rightarrow Section \\
\hline
\forall i, j : \operatorname{dom} series \bullet \\
(series\ i).route = (series\ j).route \\
\Rightarrow i = j
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Promote} \\
\Delta Global \\
\Delta Section \\
id? : Id \\
\hline
series\ id? = \theta Section \\
series' = series' \oplus \{id? \mapsto \theta Section'\}
\end{array}$$

$$GlobalTransmit \mathrel{\widehat=} \exists\, \Delta Section \bullet Promote \land Transmit$$

Then the unification techniques described above will deliver the correct combination of viewpoints if they are specified in this promotion style – namely, the syntactic inclusion of the local viewpoint – and the viewpoints are automatically consistent.

With this technique we are no longer restricted to modelling Z viewpoints at an equal level of abstraction, but we can also model the situation where one viewpoint provides the implementations of standard components to be used in another one. This is exactly the relationship that arises between a number of the ODP viewpoints. For example, the engineering viewpoint may provide standard communication components that are assumed when describing a computational viewpoint specification.

8 Related work

Several other research groups have studied partial- or viewpoint specification, in particular in Z. Ainsworth, Wallis, et al [1, 2] studied unification ("amalgamation") of Z viewpoints, using an extended notion of refinement ("co-refinement"). Daniel Jackson's work on specification by views, also in Z, uses a syntactical notion of view composition, which does not always coincide with our, more semantical, one. The approach by Frappier et al. [15] is similar to ours in the more abstract setting of homogeneous binary relations. Most of the cited papers contain small to medium examples of consistency checks within one formal method.

There are also a large number of viewpoint specification research groups in the area of requirements engineering, most of which are represented in the recent ACM FSE workshop [14]. Consistency checking, though considered essential, is not studied extensively in that context, partly due to the notations used being less formal. However, for an interesting example of viewpoint unification and inconsistency handling between an OMT object diagram viewpoint and a dataflow diagram viewpoint, cf. [9].

Various authors have reported on specification styles which use subspecifications in different languages, often with different paradigms. Most of these, however, manage to avoid the consistency issue by the use of layering techniques, or by using the various languages in different stages of development. Languages and methods that have been combined in such a way with Z include action systems [20], state charts [27], JSD and CSP [17] and SSADM [22]. However, we are not aware of any published case study of consistency checking and unification between two partial specifications in two different formal languages.

More information about our work (which is partially funded by BT Research Labs. and the EPSRC under grant number GR/K13035) can be found at:
`http://alethea.ukc.ac.uk/Dept/Computing/Research/NDS/consistency`

References

1. M. Ainsworth, A. H. Cruickshank, P. J. L. Wallis, and L. J. Groves. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, February 1994.
2. M. Ainsworth, S. Riddle, and P.J.L. Wallis. Formal validation of viewpoint specifications. *Software Engineering Journal*, 11(1):58–66, January 1996.
3. E. Boiten, J. Derrick, H. Bowman, and M. Steen. Consistency and refinement for partial specification in Z. In Gaudel and Woodcock [16], pages 287–306.
4. H. Bowman, E.A. Boiten, J. Derrick, and M. Steen. A formal theory of viewpoint consistency. Submitted for publication, 1997.
5. 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.
6. 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.
7. 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.
8. E. Cusack and G. H. B. Rafsanjani. ZEST. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 113–126. Springer-Verlag, 1992.
9. H.S. Delugach. An approach to conceptual feedback in multiple viewed software requirements modeling. In Finkelstein and Spanoudakis [14], pages 242–246.
10. J. Derrick, E.A. Boiten, H. Bowman, and M. Steen. Weak refinement in Z. In *ZUM'97*, 1997. To appear – paper available on WWW.
11. 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.
12. J. Derrick, H. Bowman, and M. Steen. Maintaining cross viewpoint consistency using Z. In K. Raymond and L. Armstrong, editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 413–424, Brisbane, Australia, February 1995. Chapman and Hall.
13. 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.
14. A. Finkelstein and G. Spanoudakis, editors. *SIGSOFT '96 International Workshop on Multiple Perspectives in Software Development (Viewpoints '96)*. ACM, 1996.
15. 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.

16. 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.
17. M.G. Hinchey. JSD, CSP and TLZ. In *Methods Integration Workshop*, Leeds, 1996.
18. ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.
19. D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4), October 1995.
20. V. Kasurinen and K. Sere. Integrating action systems and Z in a medical system specification. In Gaudel and Woodcock [16], pages 105–119.
21. 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.
22. F. Polack and K. C. Mander. Software quality assurance using the SAZ method. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 230–249. Springer-Verlag, 1994.
23. J. M. Spivey. *The Z notation: A reference manual*. Prentice Hall, 1989.
24. 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.
25. B. Strulo. How firing conditions help inheritance. In J. P. Bowen and M. G. Hinchey, editors, *Ninth Annual Z User Workshop*, LNCS 967, pages 264–275, Limerick, September 1995. Springer-Verlag.
26. R.J. van Glabbeek. The refinement theorem for ST-bisimulation semantics. In *Programming Concepts and Methods*. Elsevier Science Publishers, 1990.
27. M. Weber. Combining statecharts and Z for the design of safety-critical control systems. In Gaudel and Woodcock [16], pages 307–326.
28. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

This article was processed using the LaTeX macro package with LLNCS style