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

$[M, SPC]$

Section

$route : \text{iseq}\ SPC$
$rec, ins, sent : \text{seq}(\text{seq}\ M)$

$route \neq \langle\rangle$
$\#route = \#rec = \#ins = \#sent$
$rec = ins \frown\frown sent$
$front\ sent = tail\ rec$

InitSection

$Section'$

$\forall i : \text{dom}\ route \bullet$
$\quad rec\ i = ins\ i = sent\ i = \langle\rangle$

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:

Transmit

$\Delta Section$
$m? : M$

$route' = route$
$head\ rec' = \langle m? \rangle \frown (head\ rec)$
$tail\ rec' = tail\ rec$
$sent' = sent$

Receive

$\Delta Section$
$m! : M$

$route' = route \land rec' = rec$
$front\ ins' = front\ ins$
$last\ ins' = front(last\ ins)$
$front\ sent' = front\ sent$
$m! = last(last\ ins)$
$last\ sent' = \langle m! \rangle \frown (last\ sent)$

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:

Daemon

$\Delta Section$

$$\exists\, i : 1..\#route - 1 \mid ins\ i \neq \langle\rangle \bullet$$
$$ins'i = front(ins\ i)$$
$$ins'(i+1) = \langle last(ins\ i)\rangle \frown ins(i+1)$$
$$\forall\, j : \mathrm{dom}\, route \mid j \neq i \wedge j \neq i+1 \bullet ins'j = ins\ j$$

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: $head\ rec = in \wedge\ last\ sent = out$. These correspondences can then be documented succinctly as a relation

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

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:

$[element]$

$$null : element$$

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:

$[seq]$

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

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.

State

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

Init_State

$$\begin{array}{l} \Delta State \\ \hline s' = s_0 \\ in' = empty\_seq \\ out' = empty\_seq \end{array}$$

transmit

$\Delta State$
$ch? : element$

$s = s_0 \land s' = s_1 \land x' = ch?$

receive

$\Delta State$
$ch! : element$

$(s = s_0 \land s' = s_2 \land ch! = null) \lor$
$(in \neq out \land s = s_0 \land s' = s_3$
$\land ch! = last(in - out))$

i

$\Delta State$

$(s = s_1 \land s' = s_0 \land in' = add(x, in) \land out' = out) \lor$
$(s = s_2 \land s' = s_0 \land in' = in \land out' = out) \lor$
$(s = s_3 \land s' = s_0 \land in' = in \land out' = add(last(in - out), out))$

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.

transmit

$\Delta State$
$ch? : element$

$in' = add(ch?, in) \land out' = out$

receive

$\Delta State$
$ch! : element$

$in' = in$
$(out' = out \land ch! = null) \lor$
$(in \neq out \land ch! = last(in - out) \land$
$out' = add(ch!, out))$

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.

NState

$in, out : \operatorname{seq} M$

$out \in \operatorname{suffixes} in$

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.

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

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

- From *Section* to *NState* : $R = \lambda rec, ins, sent, route \bullet (head\ rec, last\ sent)$. 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

$[X]$

$\operatorname{suffixes} : \operatorname{seq} X \to \mathbb{P} \operatorname{seq} X$

$\forall x, y : \operatorname{seq} X \bullet y \in \operatorname{suffixes} x \Leftrightarrow rev\ y \subseteq rev\ x$

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

Un

$Section$
$NState$

$head\ \ rec = in$
$last\ \ sent = out$

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

Op

ΔS
$x? : T$

$\mathrm{pre}\ Op1 \Rightarrow Op1$
$\mathrm{pre}\ Op2 \Rightarrow Op2$

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

unReceive

ΔUn
$m! : M$

$last\ ins \neq \langle\rangle \Rightarrow Receive$
$in' = in$
$(out' = out \wedge m! = null) \vee$
$(in \neq out \wedge m! = last(in - out) \wedge out' = \langle m! \rangle \frown out)$

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

$$\begin{array}{ll} out' & out' \\ \quad = \{Un\} & \quad = \{Adreceive\} \\ last\ sent' & last(in - out) \frown out \\ \quad = \{AdReceive\} & \\ last(last\ ins) \frown last\ sent & \\ \quad = \{Un\} & \\ last(last\ ins) \frown out & \\ \quad = \{Section, Un\} & \\ last((last\ rec) - out) \frown out & \end{array}$$

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:

impreceive

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

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.