Viewpoints and Objects

John Derrick, Howard Bowman and Maarten Steen

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Phone: + 44 1227 764000, Email: {jd1,hb5,mwas}@ukc.ac.uk.)

Abstract. There have been a number of proposals to split the specification of large and complex systems into a number of inter-related specifications, called viewpoints. Such a model of multiple viewpoints forms the cornerstone of the Open Distributed Processing (ODP) standardisation initiative. We address two of the technical problems concerning the use of formal techniques within multiple viewpoint models: these are unification and consistency checking. We discuss the software engineering implications of using viewpoints, and show that object encapsulation provides the necessary support for such a model. We then consider how this might be supported by using object-oriented variants of Z.

Keywords: ODP, Viewpoints, Consistency, Object-Orientation, ZEST.

1 Introduction

With the increasing complexity of system specifications, the process of separation of concerns is being applied to the design *process* in addition to the design itself. Consideration of this and related issues has led to a number of proposals for partial specification or *viewpoints* to be used to split the specification of large and complex systems into a number of inter-related specifications, [1, 16, 12]. The purpose of decomposing a system specification into viewpoints is to enable different participants in the requirements and specification process to observe a system from a suitable perspective and at a suitable level of abstraction, [12]. One area where viewpoints are playing a prominent role is in the Open Distributed Processing (ODP) standardisation initiative, which is a natural progression from OSI, broadening the target of standardisation from the point of interconnection to the end-to-end system behaviour. The objective of ODP [12] is to enable the construction of distributed systems in a multi-vendor environment through the provision of a general architectural framework that such systems must conform to. There are five separate viewpoints presented by the ODP model: Enterprise, Information, Computational, Engineering and Technology. Requirements and specifications of an ODP system can be made from any of these viewpoints.

The ODP reference model (RM-ODP) recognises the need for formalism, with Part 4 of the RM-ODP defining an architectural semantics which describes the application of formal description techniques (FDTs) to the specification of ODP systems. Z is likely to be used for at least the information, and possibly the enterprise and computational viewpoints. The first ODP compliant specification, the Trader [13], is being written using Z for the information viewpoint.

Zave considers using partial specifications in [16] for specification of complex telephone systems. Viewpoints have also arisen in connection with requirements analysis [1]. Although the terminology used in [16, 1] is different, their work reflects a similar set of concerns to those within ODP.

Whilst it has been accepted that the viewpoint model greatly simplifies the development of system specifications and offers a powerful mechanism for handling diversity within a particular area of concern (e.g. ODP), the practicalities of how to make the approach work are only beginning to be explored. In particular, one of the consequences of adopting a multiple viewpoint approach to development is that descriptions of the same or related entities can appear in different viewpoints and must co-exist. *Consistency* of specifications across viewpoints thus becomes a central issue.

In [8] we provided a mechanism, called *unification*, to describe the combination of specifications from different viewpoints into a single specification, and to check the consistency of them. Although our motivation arises from ODP, the mechanism we describe is a general strategy for unifying two partial specifications of a system written in Z.

Although, there has been some initial work on developing techniques to combine different viewpoint descriptions or specifications, there has been little work on the software engineering consequences of adopting a multiple viewpoint approach. In particular, although the RM-ODP has adopted an object-oriented approach within viewpoints, there has been no firm evidence that object encapsulation and viewpoint specification lead to a satisfactory development scenario. Our aim here is to discuss some of the issues arising from this, and to provide initial evidence that formal approaches to object encapsulation provide the necessary support for viewpoint specification, their combination and consequent consistency checking.

Section 2 of this paper briefly sketches the unification mechanism, and in section 3 we discuss the use of consistency checking within the mechanism. These discussions are needed in order to motivate the material in sections 4 and 5. The former discusses software engineering issues which arise from the use of viewpoint specification and the latter presents a case study showing the application of these techniques to an object based approach.

2 Unification

In a model of multiple viewpoints, descriptions of the same or related entities can appear in different viewpoints and must co-exist. Clearly, the different viewpoints of the same specification must be consistent, i.e. the properties of one viewpoint specification do not contradict those of another. In addition, during the development process there must be some way to combine specifications from different viewpoints into a single implementation specification. This process of combining two specifications is known as *unification* in ODP (other terms used include amalgamation [1] and composition [16]). Because detail and design decisions in a viewpoint specification should be preserved within the complete system specification, a natural way to view unification is as the least refinement of the component viewpoints, indeed this is the approach taken in ODP, [5, 12]. Unification can also be used, because of this least refinement, as a method by which to check consistency. To check the consistency of two viewpoint specifications, we check for contradictions within the unified specification.

Unification of Z specifications depends upon the Z refinement relation, which is given in terms of two separate components - data refinement and operation refinement, [18].

Although ODP has adopted unification as being the least refinement, there are alternatives, for example [1, 2]. However, the motivation behind work described in [1, 2] comes from requirements capture, and consequently there are fundamental differences to the ODP approach. In particular, there is no requirement on unification to be a refinement (let alone the least refinement) of the individual viewpoints. Hence, the techniques developed are different from our work and cannot be adapted to provide consistency checks. Recently, the approach has been formalized in terms of augmented co-refinement [2], which is a weaker relation than the usual Z refinement relation. For the purposes of this paper we shall retain the terminology and usage derived from ODP.

The unification algorithm we describe is divided into three stages: normalisation, common refinement (which we usually term unification itself), and re-structuring. This algorithm can be shown to produce the least refinement of both viewpoints, [8].

Normalisation identifies commonality between two different viewpoint specifications, and re-writes each specification into a normal form suitable for unification. Clearly, the two specifications that are to be unified have to represent the world in the same way (e.g. if an operation is represented by a schema in one viewpoint, then the other viewpoint has to use the same name for its (possibly more complex) schema as well), and that the correspondences between the specifications have to have been identified by the specifiers involved. These will be given by mappings that describe the naming, and other, conventions in force. Once the commonality has been identified, normalisation re-names the appropriate elements of the specifications. Normalisation will also expand data-type and schema definitions into a normal form. Examples of normalisation are given in [18, 20].

Unification itself takes two normal forms and produces the least refinement of both. Because normalisation will hide some of the specification structure introduced via the schema calculus, it is necessary to perform some re-structuring after unification to re-introduce the structure chosen by the specifier.

2.1 State Unification

The purpose of state unification is to find a common state to represent both viewpoints. The state of the unification must be the least data refinement of the states of both viewpoints, since viewpoints represent partial views of an overall system description.

The essence of all constructions will be as follows. We unify declarations rather than types, so non-identical types with name clashes are resolved by re-naming, then we unify declarations as follows. If an element x is declared in both viewpoints as $x : S$ and $x : T$ respectively, then the unification will include a declaration $x : U$ where U is the least refinement of S and T. The type U will be the smallest type which contains a copy of both S and T. For example, if S and T can be embedded in some maximal type then U is just the union $S \cup T$. Given two viewpoint specifications both containing the following fragment of state description given by a schema D:

$$
\begin{array}{|l}
\hline D \\
x : S \\
\hline pred_S \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline D \\
x : T \\
\hline pred_T \\
\hline
\end{array}
$$

we unify as follows:

$$
\begin{array}{|l}
\hline D \\
x : S \cup T \\
\hline x \in S \implies pred_S \\
x \in T \implies pred_T \\
\hline
\end{array}
$$

whenever $S \cup T$ is well founded. If S and T cannot be embedded in a single type then the unification will declare x to be a member of the disjoint union of S and T, and the mechanism to describe disjoint unions has to be included in the unification. In these circumstances we again achieve the least refinement of both viewpoints. The correspondence rules will relate the types across the viewpoint specifications, and in particular will describe correspondences within the type hierarchy. Therefore the unification can construct well formed unions for overlapping types as in the example in section 5.1.

Axiomatic descriptions are unified in exactly the same manner as state schemas.

2.2 Operation Unification

Once the data descriptions have been unified, the operations from each viewpoint need to be defined in the unified specification. Unification of schemas then depends upon whether there are duplicate definitions. If an operation is defined in just one viewpoint, then it is included in the unification (with appropriate adjustments to take account of the unified state).

For operations which are defined in both viewpoint specifications, the unified specification should contain an operation which is the least refinement of both, w.r.t. the unified representation of state. The unification algorithm first adjusts each operation to take account of the unified state in the obvious manner, then combines the two operations to produce an operation which is a refinement of both viewpoint operations.

The unification of two operations is defined via their pre- and post-conditions (which can always be derived). Given two schemas A and B representing operations, both applicable on some unified state, then the unification of A and B is:

$$
\begin{array}{|l}
\hline \mathcal{U}(A, B) \\
\vdots \\
\hline pre\ A \lor pre\ B \\
pre\ A \implies post\ A \\
pre\ B \implies post\ B \\
\hline
\end{array}
$$

where the declarations are unified in the manner of the preceding subsection.

We show, in [8], that this construction is the least refinement of the two viewpoint specifications. It is also associative, allowing the natural extension of unification to an arbitrary finite number of viewpoints.

3 A word about Consistency

What is consistency? A specification is consistent if it does not contain specifications of entities which cannot possibly exist. That is, given a proof system for Z, with a validity relation $\vdash$, a specification is said to be consistent if it is not possible to prove $S \vdash \mathit{false}$. For example, a Z specification of a function will be inconsistent if the predicate part of its axiomatic definition contradicts the fact that it was declared as a function. Another way in which inconsistencies can arise in Z specifications is in the definition of free types. Examples of how such inconsistencies can occur are given in [4, 25, 22]. In general, it is undecidable whether or not a set of axioms given in a Z specification is consistent. Sufficient conditions for the consistency of certain combinations of Z paragraphs, in particular axiomatic definitions, given sets and free types, are discussed in [4].

In addition, consistency usually refers to consistency of the state model, i.e. for a given state there exists at least one possible set of bindings that satisfies the state invariant, [18, 20]. With this consistency condition comes a requirement to prove the Initialisation theorem (see below), which asserts there exists a state that satisfies the initial conditions of the model. Due to an ODP requirement associated with multiple viewpoints, we also require operation consistency, because an ODP conformance statement in Z corresponds to an operation schema(s), [24]. A conformance statement is behaviour one requires at the location that conformance is tested. Thus a given behaviour (i.e. occurrence of an operation schema) conforms if the post-conditions and invariant predicates are satisfied in the associated Z schema. That is, operations defined in two viewpoints are consistent if whenever both operations are applicable, their post-conditions are consistent (in the logical sense). Hence, operations in a unification will be implementable whenever each operation has consistent post-conditions on the conjunction of its preconditions.

Thus a viewpoint consistency check in Z involves checking the unified specification for contradictions, and has 5 components: axiom, axiomatic, state and operation consistency in addition to the Initialisation theorem. Assuming the individual viewpoints themselves are consistent, the components then take the following form.

Axiom Consistency Axioms constrain existing global constants. Hence, to check for consistency of the two viewpoints, axioms from one viewpoint have to be checked against the second viewpoint w.r.t. any terms appearing in the axioms which are defined in the second viewpoint. If an axiom contains no terms appearing in other viewpoints, its consistency checking requirements are discharged.

State Consistency Consider the general form of state unification given in Section 2.1:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{D} \\
x : S \cup T \\
\hline
x \in S \Longrightarrow pred_S \\
x \in T \Longrightarrow pred_T \\
\hline
\end{array}
$$

This state model is consistent as long as both $pred_S$ and $pred_T$ can be satisfied for $x \in S \cap T$.

Axiomatic Consistency Similar to state consistency.

Operation Consistency Consistency checking also needs to be carried out on each operation in the unified specification. The definition of operation unification means that we have to check for consistency when both pre-conditions apply. That is, if the unification of A and B is denoted $\mathcal{U}(A, B)$, we have:

$$pre\ \mathcal{U}(A, B) = pre\ A \vee pre\ B, \quad post\ \mathcal{U}(A, B) = (pre\ A \Rightarrow post\ A) \wedge (pre\ B \Rightarrow post\ B)$$

So the unification is consistent as long as the post-conditions agree whenever $(pre\ A \wedge pre\ B)$ is satisfied.

Initialisation Theorem The Initialisation Theorem is a consistency requirement of all Z specifications. It asserts that there exists a state of the general model that satisfies the initial state description, formally it takes the form:

$$\vdash \exists State \bullet InitState$$

For the unification of two viewpoints to be consistent, clearly the Initialisation Theorem must also be established for the unification. [8] discusses how this can be achieved.

4 Software Engineering Issues

The previous section elucidated the consistency checking requirements for unified viewpoint specifications. Given these requirements, it is necessary to seek software engineering strategies that make viewpoint decomposition feasible. By feasible we mean it is possible to describe viewpoints which are consistent and that the effort involved in consistency checking is minimised. In this section we explore some of the software engineering consequences of the consistency checking requirements.

Either viewpoint description and analysis will work with arbitrary specifications and specification styles; or some style guidelines or further methodology is needed for the process to become feasible. In consideration of the former position there are some issues which need to be addressed:

No encapsulation of state and operations When the state is unified, all operations acting on that state are adjusted to take account of the unified state. Hence, during unification of an operation, two adjustments are made: the first due to the unified state (declarations are updated to take account of the unified state) and the second due to the change in pre- and post-conditions. Therefore, to keep track of the consistency checking requirements the operations need to be encapsulated with the state they affect. Without this consistency checking is possible, but unrealistic for larger examples.

No operation set representation As Zave noted in [30], Z provides no means of representing the operation set of a specification (i.e. the set of operations visible by the environment). The consequences of this for unification are that if an operation schema appears in both viewpoints, then it *has* to be unified, since there is no means to tell whether it was defined (in either viewpoint) for internal structuring purposes only. If there was such information available, then internal structuring schemas could be re-named and just operations in the operational set unified.

For example, in the specification of the CME agent in section 5.1 below, the operations *Select* and *CreateSelect* are defined purely for internal structuring purposes. Given another viewpoint which contains a specification of the CME agent, we require that only the external interface represented by the *Create*, *Delete* and *Enrol* operations be unified, and that any internal operations with name clashes are resolved instead of unified.

Correspondence rules In order to describe the relation between viewpoints, the RM-ODP includes the concept of a correspondence rule. Part of their purpose is to identify the commonality between the specifications, and describe any possible renamings between them. Any viewpoint methodology will need to include mappings such as these. The limited structure in an ordinary Z specification makes a succinct naming impossible for correspondences, since, for any non-trivial systems, it is likely that a correspondence will wish to name more than one state/operation.

Viewpoint encapsulation The work of [1] indicates that in a non-object approach a large number of re-namings and re-workings of the viewpoints have to be undertaken *during* the unification process. This appears to be because the boundaries of the decomposition are not well defined, leading to viewpoint specifiers referencing and defining similar aspects of the same entity. Again this is a manifestation of the lack of encapsulation when defining the area of concern for each viewpoint specifier.

From case studies undertaken and consideration of these issues it seems that viewpoint description without any style guidelines is unlikely to be practical for anything other than small examples. Encapsulation and identity are central to the practical realization of the viewpoints model. Both of these facets could be provided by a number of software engineering methodologies, however, object-orientation is an obvious choice. Many of the problems identified above can be resolved if one adopts an object-oriented approach.

Encapsulation of state and operation The over-riding advantage of object-oriented methods is their encapsulation of state and operation. This will clearly delimit the consistency checking requirements within a unification, with each unified object generating local consistency checking requirements which do not escalate to global consistency checking problems.

Operation set representation Some, although not all, object-oriented versions of Z provide the ability to specify an operation set, or visibility list, [17, 9], which partitions all the defined operations into disjoint sets of visible and internal operations. If this is provided, then the issue of operation set representation is completely resolved. Even if such a visibility list is not provided by the language used, the encapsulation that comes with object-orientation still provides the opportunity for partial resolution of the problem.

In an object-based world it is likely that a viewpoint partitioning will include the internal specification of the behaviour of an object in only one of the viewpoints. The other viewpoints will then (possibly) reference objects from viewpoints as parameters, or place constraints on the use of those objects. Hence, in these circumstances the unification of two internal representations is unlikely to occur, and so the issue of operation set representation would not occur. Of course, if the internal specification of an object's behaviour did occur in more than one viewpoint (as in the example below), the need for a visibility list would then arise again.

Correspondence rules Identity is a key property of an object, and will allow correspondence rules to relate suitably complex parts and combination of parts of the viewpoint descriptions in a manner which is not currently supported in Z.

These considerations naturally lead to a choice of an object-based or object oriented language for viewpoint decomposition, where each viewpoint specifies a number of interacting objects. Full object-orientation is not necessarily needed, however, if it is available then object-oriented facilities such as inheritance can be exploited. It is preferable that only one viewpoint specifies the internal representation of a given object, and references to objects from one viewpoint will appear as parameters, either as inheritance within another object, or as an abstraction purely in terms of object or method names. The next section investigates the support available in Z for this approach.

5 Using Object-Oriented Techniques

The previous section indicated that an object-oriented style of specification is particularly suitable for viewpoint descriptions, and indeed the RM-ODP has adopted such an approach. We now consider the use and consequences of using viewpoints with object-oriented styles of Z specification. There have been a number of different approaches proposed for providing Z with object-oriented facilities. These include the provision of object-oriented style guidelines, and extensions to Z to allow fully object-oriented specifications. Examples of using Z in an object-oriented style include: Hall's style [10, 11]; ZERO [29]; and the ODP architectural semantics [12]. Examples of object-oriented extensions to Z include: Object-Z [9]; ZEST [7]; MooZ [17]; OOZE [3]; Schuman & Pitt [23]; and Smith [26]. See [27] for a summary and comparison of several approaches.

Z itself is not object-oriented because it does not provided sufficient support for either encapsulation or inheritance. However, it is also possible for Z to be used in an object based fashion, see [20] for a discussion, although there is nothing to keep the specifier within an object based style in contrast to the style guidelines above.

The ODP standardisation initiative requires the use of (near) standardised formal methods, hence the architectural semantics uses Z as opposed to any object-oriented variant of that language. However, given that ODP has adopted the object-oriented paradigm, there is obvious interest in object-oriented variants that can support the required ODP modelling concepts. In particular, Object-Z and ZEST are receiving attention within the ODP community as a specification medium. However, all the object-oriented extensions to Z have an unstable definition, or lack a full semantics, or both. Therefore, techniques with a flattening (or approximate flattening) into Z are of considerable interest to our work. By using such a technique we can define unification and consistency checking of viewpoints without compromising the necessity of a standardised formal description technique. Object-Z and ZEST are suitable from this perspective, however, it is unfortunate that Object-Z is moving away from a flattening semantics.

Of the Z guidelines the work of Hall is the most general. The style adds no new features to Z, however, there are conventions for writing an object-oriented specification. He also provides conventions for modelling classes and their relationships, and, in addition, there is formal support for inheritance through subtyping, [11]. In order to support encapsulation, the RM-ODP has adopted conventions for the use of Z within ODP. Here, encapsulation is achieved by letting each Z specification denote just one object. This achieves the required encapsulation, but clearly any specification of an aggregate of objects or interaction between objects cannot then be modelled within Z. There is a clear need to extend the framework offered by ODP by considering further style guidelines for the specification of collections of objects.

Here we shall show how the unification techniques can be used with ZEST for the specification of viewpoints. To do so we use ZEST to describe two viewpoints consisting of objects or aggregates of objects. We can then flatten ZEST to Z, in order to generate the unification of the two viewpoints and to check for consistency. The unification can then easily be re-assembled into a ZEST specification if further object oriented development is required. Other object-oriented variants of Z could equally have been used for the basis of this example, in particular, Object-Z would have provided a similar set of facilities as those we have called upon. Although we have applied unification by first flattening the ZEST, it is important to note that we do not lose the benefits of using object-orientation by doing this. The encapsulation can be recovered, and the consistency checking requirements still lie within the boundaries defined by the object encapsulation.

5.1 Example

The application of Z in the ODP information viewpoint to the modelling of OSI Management has been investigated by a number of researchers, see [21, 28] for introductions. We show here how the object-based approach in [28] can be used within viewpoint specifications by considering viewpoint specifications of sieve managed objects and their controlling Common Management Environment (CME) agent. We shall consider one viewpoint containing a specification of an event reporting sieve object together with a second viewpoint which describes both a sieve object and a CME agent and its manipulation of the sieve objects.

In ZEST a managed object class is described by a ZEST class, and a managed object is an instance of a managed object class. An instance of a class is created through initialisation of the class specification, which assigns values to all fixed attributes of the class. The initialisation schema provides predicates that must be satisfied on initialisation.

A managed object definition cannot include a *Create* operation, since before it is created a managed object cannot perform any operation, including *Create* itself. However, by including a *Create* operation in the CME agent viewpoint as we do below, we can describe formally the interaction between *Create* and the sieve managed object definition.

Viewpoint 1 To describe the sieve object, we first declare the types. *ObjectId* represents a set of object identifiers, and *SieveConstruct* is used in the event reporting process, its internal structure is left unspecified at this stage. The remaining types are declared as enumerated types.

$$[\mathit{SieveConstruct}, \mathit{ObjectId}]$$

$$\begin{array}{lcl}
\mathit{Operational} & ::= & \mathit{disabled} \mid \mathit{active} \mid \mathit{enabled} \mid \mathit{busy} \\
\mathit{Admin} & ::= & \mathit{locked} \mid \mathit{unlocked} \mid \mathit{shuttingdown} \\
\mathit{Event} & ::= & \mathit{nothing} \mid \mathit{enrol} \mid \mathit{deenrol} \\
\mathit{Status} & ::= & \mathit{created} \mid \mathit{deleted}
\end{array}$$

Status models the life-cycle of the sieve object, and is used as an internal mechanism to control which operations are applicable at a given point within an object's existence.

The ZEST sieve class is then defined by:

Sieve

$$\begin{array}{l}
\mathit{sieveid} : \mathit{ObjectId} \\
\mathit{destadd} : \mathit{ObjectId}
\end{array}$$

$$\begin{array}{l}
\mathit{opstate} : \mathit{Operational} \\
\mathit{sico} : \mathit{SieveConstruct} \\
\mathit{adminstate} : \mathit{Admin} \\
\mathit{status} : \mathit{Status}
\end{array}$$

INIT

$$\begin{array}{l}
\mathit{opstate} = \mathit{active} \\
\mathit{adminstate} = \mathit{unlocked}
\end{array}$$

$$\mathit{filter} : \mathit{Event} \leftrightarrow \mathit{SieveConstruct}$$

Filter

$$\begin{array}{l}
\mathit{event}? : \mathit{Event} \\
\mathit{notification}! : \mathit{Event} \\
\hline
\mathit{status} \neq \mathit{deleted} \\
\mathit{opstate} = \mathit{active} \land \mathit{adminstate} = \mathit{unlocked} \\
(\mathit{event}?, \mathit{sico}) \in \mathit{filter} \Rightarrow \mathit{notification}! = \mathit{event}? \\
(\mathit{event}?, \mathit{sico}) \notin \mathit{filter} \Rightarrow \mathit{notification}! = \mathit{nothing}
\end{array}$$

Within the class definition we have described just one of the operations available within a sieve object (for a full description of operations see [21]). The relation *filter* represents criteria to decide which events to filter out and which to pass on and *Filter* represents the operation to perform the filtering.

Viewpoint 2 The second viewpoint contains a description of a controlling CME agent together with a sieve object. For our purposes here we present a very simplified version of an agent which consists of a number of sieve managed objects. The CME agent promotes the *Delete* operation defined on individual sieve objects, and defines a *Create* operation to instantiate sieve objects as required. First of all we declare the types

$$
\begin{array}{lll}
[SieveConstruct, ObjectId] & & \\
Operational & ::= & disabled \mid active \mid enabled \mid busy \\
Admin & ::= & locked \mid unlocked \mid shuttingdown \\
Event & ::= & nothing \mid enrol \mid deenrol \\
Status & ::= & being\_created \mid created \mid deleted
\end{array}
$$

Notice that in this viewpoint *Status* includes an additional value, *being_created*.

The ZEST sieve object is then specified in this viewpoint as shown in Figure 1.

Within this class definition an operation to delete a sieve is declared. Upon deletion a sieve sends a *deenrol* notification to its environment, and moves into a state where no further operations can be applied. A *CMEagent* is modelled as a collection of sieve objects, where and initially no sieve objects have been created.

CMEagent

$id : ObjectId$

$sieves : \mathbb{P}\, Sieve$

INIT

$sieves = \varnothing$

Select

$s? : Sieve$

$s? \in sieves$

CreateSelect

$\Delta(sieves)$
$s? : Sieve$

$s? \notin sieves$
$sieves' = sieves \cup \{s?\}$

$Delete \mathrel{\widehat{=}} Select \bullet s?.Delete$
$Enrol \mathrel{\widehat{=}} Select \bullet s?.Enrol$
$Create \mathrel{\widehat{=}} CreateSelect \bullet s?.Init$

Notice that the extra structuring available in ZEST does away with the need to define framing schemas to achieve the promotion of operations from Sieve to the aggregate of sieves used in the agent. The agent operation to delete a specific sieve object can now be defined quite simply by referencing the method of the appropriate sieve object. The other managed object operations are promoted in a similar fashion. It is situations such as these where a visibility list is necessary, because we do not want operations *Select* and *CreateSelect* to be visible to the environment. Ideally they should be declared as internal operations. Recently, a small extension to ZEST included such a mechanism.

Notice that the *Create* operation is not part of the sieve specification, so we have preserved the concept that *Create* must occur before any operation in the sieve specification can be applied.

Unification of Viewpoints To describe the unification of viewpoints, we decorate with subscripts, so for example $Filter_1$ is the schema $Filter$ from the first viewpoint. To apply the unification of the two viewpoints we first flatten both ZEST specifications into Z specifications. The flattening used is that defined in [19], and we do not repeat the details here. However, the flattening of the CMEagent need not be calculated because no other viewpoint contains a specification of the CMEagent's behaviour. Therefore it suffices to consider just the flattening of the ZEST sieve specifications. We give this here, and then unify the two Z specifications, however, experience shows that it is straightforward to factor the unification through the ZEST extended structuring in an obvious manner, and that the unification can be derived just as easily from the ZEST specifications.

The flattening of the sieve object in the first viewpoint is:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sieve}} \\
\hline
sieveid : ObjectId \\
destadd : ObjectId \\
opstate : Operational \\
sico : SieveConstruct \\
adminstate : Admin \\
status : Status \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitSieve}} \\
\hline
Sieve \\
\hline
opstate = active \\
adminstate = unlocked \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
filter : Event \leftrightarrow SieveConstruct
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Filter}} \\
\hline
\Xi Sieve \\
event? : Event \\
notification! : Event \\
\hline
status \neq deleted \\
opstate = active \land adminstate = unlocked \\
(event?, sico) \in filter \Rightarrow notification! = event? \\
(event?, sico) \notin filter \Rightarrow notification! = nothing \\
\hline
\end{array}
$$

The equivalent Z sieve object in the second viewpoint can be derived in a similar fashion. To unify the two specifications we first unify the state. The only conflict in the declarations are due to differing types $Status_1$ and $Status_2$. To resolve this conflict, the type $Status$ in the unification is taken as the least refinement of $Status_1$ and $Status_2$ (i.e. $Status_1 \cup Status_2$, and this is well formed because the correspondence rules collapse the type hierarchy), and state unification is applied to the schema *Sieve*. Hence, in addition to the declarations which are not in conflict, the unification will contain the following:

$$Status ::= being\_created \mid created \mid deleted$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Sieve\;}} \\
sieveid : ObjectId \\
destadd : ObjectId \\
opstate : Operational \\
sico : SieveConstruct \\
adminstate : Admin \\
status : Status \\
\hline
status \in \{created, deleted\} \Rightarrow true \\
status \in \{being\_created, created, deleted\} \Rightarrow \\
\qquad (opstate \in \{active, disabled\} \land adminstate \in \{locked, unlocked\}) \\
\hline
\end{array}
$$

Upon simplification the schema *Sieve* becomes:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Sieve\;}} \\
sieveid : ObjectId \\
destadd : ObjectId \\
opstate : Operational \\
sico : SieveConstruct \\
adminstate : Admin \\
status : Status \\
\hline
opstate \in \{active, disabled\} \\
adminstate \in \{locked, unlocked\} \\
\hline
\end{array}
$$

In a similar fashion we unify $InitSieve_1$ and $InitSieve_2$, which simplifies to:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;InitSieve\;}} \\
Sieve \\
\hline
status = being\_created \\
opstate = active \\
adminstate = unlocked \\
\hline
\end{array}
$$

The *Delete* and *Enrol* schemas are defined in just one viewpoint. Hence, both these schemas are included in the unification (with adjustments due to the unified state schema *Sieve*). Similarly the unification contains both relations *filter* and *newfilter*.

To unify $Filter_1$ and $Filter_2$ we first adjust $Filter_1$ due to the unified state schema. The predicate part of $Filter_1$ is then

$$\begin{array}{l} status \in \{created, deleted\} \Rightarrow \\ \quad (status \neq deleted \land opstate = active \land adminstate = unlocked) \end{array}$$

Calculation of the pre-conditions $preFilter_1 \lor preFilter_2$ then simplifies to:

$$(status = created \land opstate = active \land adminstate = unlocked)$$

Thus the unification of $Filter_1$ and $Filter_2$ is then given by:

$$\begin{array}{|l} \multicolumn{1}{l}{\_\_ Filter \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\ \Xi Sieve \\ event? : Event \\ notification! : Event \\ \hline status = created \land opstate = active \land adminstate = unlocked \\ (event?, sico) \in filter \Rightarrow notification! = event? \\ (event?, sico) \notin filter \Rightarrow notification! = nothing \\ (event?, sico) \notin newfilter \Rightarrow notification! = nothing \\ \hline \end{array}$$

Since we were unifying two Sieve objects, the unification can easily be re-written into the equivalent ZEST class shown in Figure 2.

5.2 Conclusions from Case Studies

The above example illustrates the use of object specifications within viewpoints and their unification. Although the example is simple, it clearly shows how viewpoints can reference other objects and their methods, or contain a partial description of an object's behaviour. We have undertaken a number of case studies in order to test the hypothesis that object-oriented description is the preferable viewpoint specification medium, and our conclusions so far support this claim. The studies involving non-object based descriptions were significantly harder to check for consistency and much harder to specify in an independent fashion in the viewpoints, the specification in [1] is another indication of the difficulty of non-object based viewpoint specifications.

Conversely, the object based viewpoint descriptions were much more successful. When the viewpoints contain only references to objects defined in other viewpoints (as opposed to specifying any of their behaviour) consistency checking is relatively straightforward (although the viewpoints can still be inconsistent). If two viewpoints both contain (partial) descriptions of the same object, then there can be a non-trivial consistency checking process, however, due to encapsulation the boundaries that inconsistency can arise within are well defined.

Examples were undertaken using the following styles: Hall's conventions; encapsulation as defined by the ODP architectural semantics; ZEST and finally using Z specifications produced from the object-oriented methodology described by Smith [26]. The style of the object-oriented variant chosen did not significantly affect the success or otherwise of the viewpoint specification or unification. There are clear merits in using Z without extended syntax, particularly in the use within ISO initiatives. To that extent, there are clear advantages in using the work of Hall and Smith. Hall in particular offers formal and well-defined support for inheritance, which is lacking for some other Z object-oriented variants. The extended syntax approaches have advantages for the developer, who is then not constrained by conventions for embedding object-orientation in Z, but only if a clear semantics, and preferably a flattening into Z, can be given.

5.3 Relation between Unification and Inheritance

It is important to recognise that unification is a 'horizontal' rather than 'vertical' development activity. By that we mean it is used to check development at a particular stage (consistency checking) or possibly to combine development specifications (unification), rather than a development activity that serves to define the implementation more closely (as in refinement or inheritance). Given that unification is a horizontal activity, one needs to describe the relationship between it and vertical development activities. The relationship between unification and refinement is well known since unification is based upon (least) refinement. We describe here the relation between inheritance and unification.

To do so we need a formal approach to inheritance and subtyping. In [11], Hall discusses known definitions in terms of both extensional and intensional semantics. Given that we are interested in behaviour of specifications, we shall consider definitions due to intensional semantics here. His intensional meaning of subclass is in terms of subclass instances being valid implementations of the superclass, however, the definition is different from a refinement relation (as one would expect). To exhibit subtyping there must exist a retrieve relation *Abs* between the superclass and subclass such that the following are true.

S1 $\forall Superstate; \ Substate \bullet \text{pre}\, Superop \land Abs \Rightarrow \text{pre}\, Subop$

S2 $\forall Superstate; \ Substate; \ Substate' \bullet \text{pre}\, Superop \land Abs \land Subop \Rightarrow$
$(\exists Superstate' \bullet Abs' \land Superop)$

S3 $\forall Substate \bullet (\exists Superstate \bullet Abs)$

Only the third rule differs from the rules for refinement, see [11] for justification of this. Hall also compares his definition with those of Cusack [6], Lano & Haughton [14] and Liskov & Wing [15]. We are interested here in the relation between unification and the individual viewpoints. In these circumstances the retrieve relations will be partial functions (and only total if one viewpoint is degenerate). In this case Hall's subtyping implies subtyping in the sense of Cusack, Lano & Haughton and Liskov & Wing (ignoring the history predicates of the latter two). In particular, the rules S1-3 suffice for subtyping in both Hall and ZEST, and we will thus work with this definition.

It is easy to construct examples to show that the unification of two viewpoints is not in general a subtype of each viewpoint. However, this is unsurprising because one viewpoint is only a partial description of an object's behaviour. Instead the natural result to seek is the following:

Theorem 1. *Let P_i, O_i be objects in viewpoint i. Let P_i be a subtype of O_i. Then $\mathcal{U}(P_1, P_2)$ is a subtype of $\mathcal{U}(O_1, O_2)$, where $\mathcal{U}$ is the unification operator between viewpoints.*

Proof

The full proof involves construction of appropriate retrieve relations between $\mathcal{U}(P_1, P_2)$ and $\mathcal{U}(O_1, O_2)$ in a manner similar to the proof that unification is the least refinement, see [8]. The outline of the proof is as follows:

The subtyping rules S1 and S2 between $\mathcal{U}(P_1, P_2)$ and $\mathcal{U}(O_1, O_2)$ are satisfied because unification is the least refinement.

For S3, note that every state in the $\mathcal{U}(O_1, O_2)$ unification appears in either O_1 or O_2 or both. Thus every state in $\mathcal{U}(P_1, P_2)$ is related to some state in $\mathcal{U}(O_1, O_2)$ via the retrieve relation defined for the least refinement. □

It is straightforward to construct examples to show the converse is not true, that is $\mathcal{U}(P_1, P_2)$ being a subtype of $\mathcal{U}(O_1, O_2)$ does not imply that P_i is a subtype of O_i.

The theorem then provides a sound footing for the use of object-oriented techniques in viewpoint descriptions. The relationship between unification and multiple inheritance is clearly of importance (especially w.r.t method consistency), and is currently under investigation.

6 Conclusion

Formal description techniques are being employed extensively in ODP and have proved valuable in supporting precise definition of reference model concepts. The use of viewpoints to enable separation of concerns to be undertaken at the specification stage is a cornerstone of the ODP model. Therefore two issues of importance to ODP and other models of multiple viewpoints are unification and consistency checking.

We discussed the need for encapsulation of the consistency checking boundaries within a unification, and showed how object-oriented methodologies provide the necessary support for such a process. This was illustrated by providing a unification mechanism on an object-oriented variant of Z, where we specified a number of viewpoints of an OSI Management application.

Acknowledgements

This work was partially funded by British Telecom Labs., Martlesham, Ipswich, UK and by the Engineering and Physical Sciences Research Council under grant number GR/K13035. Thanks to David Wolfram for discussing the semantics of ZEST with the authors.

References

1. M Ainsworth, AH Cruickshank, LJ Groves, and PJL Wallis. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, February 1994.
2. M Ainsworth and PJL Wallis. Co-refinement. In D Till, editor, *Proc. 6th Refinement Workshop*, City University, London, 5th–7th January 1994. Springer-Verlag.

3. A. J. Alencar and J. A. Goguen. OOZE: An object oriented Z environment. In P. America, editor, *ECOOP '91 - Object-Oriented Programming*, LNCS 512, pages 180–199. Springer-Verlag, 1991.
4. R. D. Arthan. On free type definitions in Z. In J. E. Nicholls, editor, *Sixth Annual Z User Workshop*, pages 40–58, York, December 1991. Springer-Verlag.
5. G. Cowen, J. Derrick, M. Gill, G. Girling (editor), A. Herbert, P. F. Linington, D. Rayner, F. Schulz, and R. Soley. *Prost Report of the Study on Testing for Open Distributed Processing*. APM Ltd, 1993.
6. E. Cusack. Inheritance in object oriented Z. In P. America, editor, *ECOOP '91 - Object-Oriented Programming*, LNCS 512, pages 167–179. Springer-Verlag, 1991.
7. 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.
8. J. Derrick, H. Bowman, and M. Steen. Maintaining cross viewpoint consistency using Z. In *IFIP International Conference on Open Distributed Processing*. Chapman Hall, 1995.
9. R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language version 1. Technical Report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, May 1991.
10. A. J. Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90 VDM and Z - Formal Methods in Software Development*, LNCS 428, pages 290–318, Kiel, FRG, April 1990. Springer-Verlag.
11. J. Hall. Specifying and interpreting class hierarchies in Z. In J. Bowen and J. Hall, editors, *Eighth Annual Z User Workshop*, pages 120–138, Cambridge, July 1994. Springer-Verlag.
12. ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.
13. ITU/ISO CD ISO 13235/ITU.TS Rec.9tr. *ODP Trading Function*, 1994.
14. K. Lano and H. Haughton. Reuse and adaption of Z specifications. In J. P. Bowen and J. E. Nicholls, editors, *Seventh Annual Z User Workshop*, pages 62–90, London, December 1992. Springer-Verlag.
15. B. Liskov and J. M. Wing. A new definition of the subtype relation. In O. M. Nierstrasz, editor, *ECOOP '93 - Object-Oriented Programming*, LNCS 707, pages 118–141. Springer-Verlag, 1993.
16. P. Mataga and P. Zave. Formal specification of telephone features. In J. Bowen and J. Hall, editors, *Eighth Annual Z User Workshop*, pages 29–50, Cambridge, July 1994. Springer-Verlag.
17. S. L. Meira and A. L. C. Cavalcanti. Modular object oriented Z specifications. In J. E. Nicholls, editor, *Fifth Annual Z User Workshop*, pages 173–192, Oxford, December 1990. Springer-Verlag.
18. B. Potter, J. Sinclair, and D. Till. *An introduction to formal specification and Z.* Prentice Hall, 1991.
19. G. H. B. Rafsanjani. ZEST - Z Extended with Structuring: A users's guide. Technical report, BT, June 1994.
20. B. Ratcliff. *Introducing specification using Z.* McGraw-Hill, 1994.
21. S. Rudkin. Modelling information objects in Z. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 267–280, Berlin, Germany, September 1991. North-Holland.
22. M. Saaltink. Z and Eves. In J. E. Nicholls, editor, *Sixth Annual Z User Workshop*, pages 223–243, York, December 1991. Springer-Verlag.

23. S. A. Schuman, D. H. Pitt, and P. J. Byers. Object-oriented process specification. In C. Rattray, editor, *Specification and Verification of Concurrent Systems*, Workshops in Computing, pages 21–70. Springer-Verlag, 1990.
24. 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.
25. A. Smith. On recursive free types in Z. In J. E. Nicholls, editor, *Sixth Annual Z User Workshop*, pages 3–39, York, December 1991. Springer-Verlag.
26. G. Smith. An object-oriented development framework for Z. In J. Bowen and J. Hall, editors, *Eighth Annual Z User Workshop*, pages 89–107, Cambridge, July 1994. Springer-Verlag.
27. S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, 1992.
28. C. Wezeman and A. J. Judge. Z for managed objects. In J. Bowen and J. Hall, editors, *Eighth Annual Z User Workshop*, pages 108–119, Cambridge, July 1994. Springer-Verlag.
29. P. J. Whysall and J. A. McDermid. An approach to object oriented specification using Z. In J. E. Nicholls, editor, *Fifth Annual Z User Workshop*, pages 193–215, Oxford, December 1990. Springer-Verlag.
30. P. Zave and M. Jackson. Techniques for partial specification and specification of switching systems. In J. E. Nicholls, editor, *Sixth Annual Z User Workshop*, pages 205–219, York, December 1991. Springer-Verlag.

Sieve

$sieveid : ObjectId$
$destadd : ObjectId$

$opstate : Operational$
$sico : SieveConstruct$
$adminstate : Admin$
$status : Status$

$opstate \in \{active, disabled\}$
$adminstate \in \{locked, unlocked\}$

INIT

$status = being\_created$
$opstate = active$
$adminstate = unlocked$

Delete

$\Delta(status)$
$notification! : Event$

$status = created$
$notification! = deenrol$
$status' = deleted$

Enrol

$\Delta(status)$
$notification! : Event$

$status = being\_created$
$status' = created$
$notification! = enrol$

$newfilter : Event \leftrightarrow SieveConstruct$

Filter

$event? : Event$
$notification! : Event$

$status = created$
$opstate = active \wedge adminstate = unlocked$
$(event?, sico) \notin newfilter \Rightarrow notification! = nothing$

Fig. 1. ZEST sieve object specification

Sieve

$sieveid : ObjectId$
$destadd : ObjectId$

$opstate : Operational$
$sico : SieveConstruct$
$adminstate : Admin$
$status : Status$

$opstate \in \{active, disabled\}$
$adminstate \in \{locked, unlocked\}$

INIT

$status = being\_created$
$opstate = active$
$adminstate = unlocked$

Enrol

$\Delta(status)$
$notification! : Event$

$status = being\_created$
$status' = created$
$notification! = enrol$

Delete

$\Delta(status)$
$notification! : Event$

$status = created$
$notification! = deenrol$
$status' = deleted$

$filter, newfilter : Event \leftrightarrow SieveConstruct$

Filter

$event? : Event$
$notification! : Event$

$status = created \wedge opstate = active \wedge adminstate = unlocked$
$(event?, sico) \in filter \Rightarrow notification! = event?$
$(event?, sico) \notin filter \Rightarrow notification! = nothing$
$(event?, sico) \notin newfilter \Rightarrow notification! = nothing$

Fig. 2. Equivalent ZEST class