A Case Study in Partial Specification: Consistency and Refinement for Object-Z

Chris Taylor, John Derrick and Eerke Boiten
Computing Laboratory, University of Kent,
Canterbury, CT2 7NF, UK
C.N.Taylor-1,J.Derrick,E.A.Boiten@ukc.ac.uk

Abstract

The 'viewpoint' approach, in which a system is described by several partial specifications, has been proposed as a way of making complex computing systems more understandable. The ISO's Open Distributing Processing (ODP) framework is an architecture for open distributed systems, involving five named viewpoints. This paper compares two partial specifications of a lending library — from the ODP's Enterprise and Information Viewpoints — and discusses the relation between them. Both specifications are written in Object-Z, an object-oriented variant of Z. Examining how such partial specifications might be unified raises broader issues of refinement and mutual consistency of partial specifications in Object-Z.

1. Introduction

Distributed computing systems, consisting of multiple interacting software and hardware components, often physically distributed across networks, are now commonplace. Their increasing complexity motivates an approach in which systems are specified from several different *viewpoints* [4] — each a partial specification, focusing on a particular aspect.

The ISO's Open Distributing Processing (ODP) framework, an architecture proposed for open distributed systems [6], identifies five named viewpoints, as follows:

- *Enterprise Viewpoint* — focuses on the overall scope, purpose, and policies of the system.
- *Information Viewpoint* — specifies in a fairly abstract way the information involved in the system, and how it is processed, without describing the distributed architecture that will be used.
- *Computational Viewpoint* — a functional decomposition of the system into objects that interact via specific interfaces.
- *Engineering Viewpoint* — a specification of the mechanisms and functions needed to support interaction between the distributed objects of the system.
- *Technology Viewpoint* — concerned with the concrete technological infrastructure of a system, i.e. the particular hardware and software components involved, and how they are interrelated.

ODP does not prescribe particular specification formalisms, software, or hardware. The viewpoints are informally defined in natural language — albeit at much greater length than given above — and so are inevitably somewhat vague.

In the viewpoint method, the problem arises of relating different viewpoint specifications — which may not even be in the same language — and ensuring that they are 'consistent', in some sense (see [3]). One approach is to say that partial specifications are mutually consistent if and only if they have a *common refinement*. Finding such a refinement is sometimes referred to as *unification*. The problem then becomes one of defining appropriate refinement rules which preserve certain properties. Another issue important in practice as regards viewpoints — although not addressed in this paper — is tool support (see [4]).

Our work on using the specification language Z (see [9]) for partial specification has led to new insights into refinement. A variety of refinement relations have been defined (see [2]) to account for specifications being 'partial' in a number of different senses. For example, a specification can be partial in describing only a certain aspect of the behaviour (e.g. a state-space specification without timing constraints), only a subset of the possible operations, only a certain perspective (e.g. an external user's), only a certain level of implementation, or only a certain subsystem of the whole.

Object-Z [8] is one of the best developed of several Z-like object-oriented specification languages. In comparison to the work on Z refinement, there has been little work on Object-Z refinement, particularly as regards partial or viewpoint specification. This paper examines some of the issues involved, by summarising a case study in which two Object-Z specifications of a lending library are compared (see [11]). The first (in Section 2) is from the Enterprise Viewpoint. It was derived using a policy specification language, and a set of translation rules from that language to Object-Z (see [10]). The second (in Section 3) is from the Information Viewpoint.

The purpose of the case study was twofold. One aspect was to consider to what extent partial specification can be successful: thus in our case study, different project members were responsible for the different viewpoint specifications, with little communication between them. We wanted to see if it was possible to reconcile the resulting differences. Section 4 compares the viewpoint specifications and comments on their differences. The second aspect was to look at what general conclusions can be drawn, regarding the refinement of partial specifications in Object-Z. Section 5 uses the case study to suggest possible refinement rules for partial specification in Object-Z. Section 6 concludes the paper.

2. An Enterprise Viewpoint library specification

This section outlines the Enterprise Viewpoint library specification, given in full in [10]. The informal ODP definition of the Enterprise Viewpoint refers to describing a system as a 'community' with an overall objective, consisting of entities that play various 'roles', some as 'actors' (agents), others as 'artifacts' (passive objects). The roles are constrained by 'policies', i.e. permissions, obligations, and prohibitions. In the library case, the objective is to share the collection fairly and efficiently amongst the members, and the policies relate to loans. For example, the regulation *Academic staff may borrow 25 books or periodicals ...* denotes a permission, and the regulation *Items borrowed must be returned by the due day and time* denotes an obligation.

To formalise such policies, [10] develops a simple logic-based policy language, with a mapping into Object-Z. This helps to support consistency checking between different viewpoints, since Object-Z has been suggested as a notation for the Information Viewpoint. However, this paper only describes the Object-Z form of the Enterprise specification.

2.1. Object-Z

An Object-Z specification includes several *class schemas*, each corresponding to an ADT, of which one represents the type of system being modelled, e.g. in our case study, the *Library* class. Variable declarations such as $x : ClassName$ are allowed, where x denotes an unique identifier for, or pointer to, an object of the class *Classname* (the notation $x :\downarrow ClassName$ is also used, to mean that x points to an object of type *ClassName* or of any of its descendant classes). If *Op* is an operation of class *Classname*, the notation $x.Op$ represents the execution of *Op* on the object to which x refers. For example, in one of the library specifications given, the *Library* class has an attribute *clock* of class *Clock*, and the expression $clock.Tick$ represents the execution on the library's clock of its own (lower-level) *Tick* operation.

Each operation has a Δ-list, showing the attributes it is allowed to change — attributes not in the Δ-list are not changed by that operation (unless they are 'secondary' attributes, separated from the main attributes by a Δ symbol, in which case they may change in any way consistent with preserving the state invariant). The names of the visible operations of a class are shown bracketed after the $\upharpoonright$ symbol, at the top of the class schema, and the corresponding operation schemas appear below the state schema.

The ‘firing’ interpretation of operations is assumed in Object-Z, i.e. an operation cannot occur unless its pre-condition is satisfied. This contrasts with an interpretation often assumed in standard Z, in which operations are viewed as total relations on the state space, such that pre-states which do not satisfy the pre-condition are related to any post-state.

2.2. Component object types

Date, *Time*, and *Money* are given types. The current date and time are represented by the special globally declared terms $today : Date$ and $now : Time$, assumed to have changing rather than fixed values.

Enterprise Viewpoint roles are represented by classes. *Item* and *Loan* are passive ‘artifact’ roles. The *Item* class (outlined below) has operations *CheckIn* and *CheckOut*. *Book* and *Periodical* are subclasses of *Item*. The *Loan* class has attributes *borrower*, *item*, *issue_date*, and *due_date*, and no operations.

$$
\begin{array}{l}
\underline{\ Item\ } \\
\upharpoonright(CheckOut, CheckIn) \\
\quad onloan : \mathbb{B} \\
\quad loan : \mathbb{P}\, Loan \\
\quad \overline{\#loan \leqslant 1 \land onloan \Leftrightarrow loan \neq \varnothing} \\
\quad \vdots
\end{array}
$$

Borrower is an ‘actor’ role. The *UpdateFines* operation is not in the visibility list, which is taken to mean that it occurs as soon as its precondition is true, without interaction from the environment. The subclasses of *Borrower* — *ACBorrower*, *PGBorrower*, and *UGBorrower* — use different preconditions for the *Borrow* operation and different values of the *allowance* attribute to express the policies concerning borrowing.

$$
\begin{array}{l}
\underline{\ Borrower\ } \\
\upharpoonright(Borrow, Return, PayFine) \\
\quad allowance : \mathbb{N} \\
\quad fines : Money \\
\quad loans : \mathbb{P}\, Loan \\
\quad \overline{\#loans \leqslant allowance} \\
\quad \underline{\ Borrow\ } \\
\quad \quad \ldots \\
\quad \vdots \\
\quad \underline{\ UpdateFines\ } \\
\quad \quad \Delta(fines) \\
\quad \quad \overline{\ldots}
\end{array}
$$

The librarian’s role is to prevent unauthorised loans and to maintain the loan records. Librarians may issue and return items or receive fines.

$$
\begin{array}{l}
\underline{\ Librarian\ } \\
\upharpoonright(Issue, Return, ReceiveFine) \\
\quad \underline{\ Issue\ } \\
\quad \quad borrower? : Borrower \\
\quad \quad item? : Item \\
\quad \quad \overline{\#(borrower?.loans)} \\
\quad \quad \quad < borrower?.allowance \\
\quad \quad \vdots \\
\quad \vdots
\end{array}
$$

2.3. The library community

The library community consists of borrowers, items and librarians. Its actions (*Borrow*, *Return*, and *PayFine*) are interactions between two or three different roles. Additional constraints on the borrow and return actions ensure that appropriate loans are created or destroyed.

Library

$$\upharpoonright(\mathit{Borrow}, \mathit{Return}, \mathit{PayFine})$$

$$\begin{array}{l}
\mathit{borrowers} : \mathbb{F}\,\mathit{Borrower} \\
\mathit{items} : \mathbb{F}\,\mathit{Item} \\
\mathit{librarians} : \mathbb{F}\,\mathit{Librarian} \\
\mathit{loans} : \mathbb{F}\,\mathit{Loan} \\
\hline
\forall\,\mathit{loan} : \mathit{loans} \bullet \\
\quad \mathit{loan.borrower} \in \mathit{borrowers} \\
\quad \wedge\ \mathit{loan.item} \in \mathit{items} \\
\quad \wedge\ \mathit{loan} \in \mathit{loan.borrower.loans} \\
\quad \wedge\ \mathit{loan.item.loan} = \{\mathit{loan}\} \\
\vdots
\end{array}$$

$$\vdots$$

3. An Information Viewpoint library specification

This section outlines an Object-Z Information Viewpoint specification of the library system, given in full in [11]. It is a high-level functional specification, defining at a fairly abstract level the information needed in the system state, and the operations required to manipulate it. Whereas the Enterprise Viewpoint concentrated on high-level policies, this viewpoint looks more closely at the stock of the library, its collections and users, and the fines on overdue loans. The outline given here illustrates the scope of the Information Viewpoint and the level of detail it might typically include, although given the informal nature of the ODP framework, it is difficult to be very precise about this.

3.1. Component object types

The library stock consists of 'items', i.e. actual physical books, periodicals, etc. Each item has an associated 'abstract item', i.e. the abstract work of which it is a particular copy. 'Holders' are entities that can 'hold' items, in an abstract sense. The given type *Holder* is partitioned into users (i.e. people), library loan collections, and 'other holders' (which includes objects used to represent a library store, a reference collection, and a 'disposal bin').

The *Item* class has attributes representing the related abstract item, the category (book or periodical), the 'location' (current holder of that item), and the 'home' (holder to which that item 'belongs').

Item

$$\upharpoonright(\mathit{MoveItem}, \mathit{ChangeItemHome})$$

$$\begin{array}{l}
\mathit{abitem} : \mathit{AbItem} \\
\mathit{cat} : \mathit{Cat} \\
\mathit{loc}, \mathit{home} : \mathit{Holder} \\
\hline
\mathit{cat} = \mathit{abitem.cat}
\end{array}$$

$$\vdots$$

The *MoveItem* operation changes the location, but not the home; whereas the *ChangeItemHome* operation changes both, by moving the item from its home to a new current location, which becomes the new home as well. These two operations are used to define *Library* class operations that transfer items between 'holders' — some of which change the library stock (e.g. buying and selling of items, disposal of items), whereas as others (e.g. issuing and returning) do not.

Because the behaviour in our case study is heavily time-dependent, our Information Viewpoint specification models time explicitly. However, other Information Viewpoint descriptions might not need to do this. Here we use a *Clock* class, where 'clocks' measure both the time and the date. The primary attributes of a clock are its start date and time, and its tick count, incremented by the *Tick* operation.

A *Loan* class is defined, without operations. Its attributes are: an identity number, the item and user involved, the issue date and time, the due date, the loan collection from which the loan is made, and the item category and abstract item of the physical item involved.

A library user may join, leave, or change membership type whilst a member. Three global functions express borrowing restrictions, mapping each user type to a maximum total number of loans, a maximum number for each item category, and a level of fines at which further borrowing is blocked.

We can see clearly now the need for consistency checking between the viewpoints. Borrowing operations are part of the Information Viewpoint, but also occurred in the policies in the Enterprise Viewpoint. Clearly the two specifications have to be consistent in any such areas of overlap.

3.3. Fines

Fines are expressed as natural numbers. A user's net fines are the cumulative fines incurred, minus those paid. The fines incurred are the sum of those on current loans and those on old (i.e. returned) loans. A state invariant specifies that the fine on any current loan is zero if the loan's due date is after the library's clock date; and otherwise, is equal to the number of days that the loan is overdue times the charge per day. Every library operation 'ticks' the clock, so its date periodically advances.

3.4. The library class

The top-level library class has primary attributes such as the stock of items, the main and short-term loan collections, the set of all loans so far, and the total fines paid by each user.

Lib

$\upharpoonright (Tick, AddMember, ...)$

$clock : Clock$
$stock : \mathbb{F}\, Item$
$main, short : LoanColl$
$status : User \rightarrow Status$
$loans\_so\_far, old\_loans : \mathbb{F}\, Loan$
$fines\_paid : User \rightarrow \mathbb{N}$
$return\_time : Loan \nrightarrow Time$
$\ldots$
Δ
$loans\_out : \mathbb{F}\, Loan$
$current\_fine : Loan \nrightarrow \mathbb{N}$
$net\_fines : User \rightarrow \mathbb{N}$

$\ldots$

$\textsc{Init}$
$clock.\textsc{Init}$
$loans\_so\_far = \varnothing$
$\vdots$

$\vdots$

Secondary attributes are also used, e.g. the set of loans currently out. A state invariant and initialisation are specified. The operations in this class represent the Information Viewpoint behaviour: for example, there are operations *AddMember*, *PayFine*, *Issue*, etc. The overlap with the Enterprise Viewpoint is again clear, since the library class also models paying fines, borrowing, etc.

4. Comparison of the specifications

This section compares the IVS (Information Viewpoint Specification) and EVS (Enterprise Viewpoint Specification). Many of the issues raised relate to the general problem of unifying partial specifications in Object-Z: indeed, this case study addresses broader questions about Object-Z refinement, as well as questions relevant to ODP.

4.1. Differences between the specifications

Table 1 summarises the kinds of entity used in the IVS and EVS. Subclass types are indented with respect to their parent class type, as are named subsets of a given type. Roughly corresponding types are horizontally aligned, with a gap being left if there is no corresponding type.

Table 1: Comparison of IVS and EVS types

IVS	EVS
Library(C:18att:23ops)	*Library*(C:4att:3ops)
Item(C:4att:2ops)	*Item*(C:2att:2ops)
	– *Book*(as above)
	– *Periodical*(as above)
Loan(C:9att:0ops)	*Loan*(C:4att:0ops)
Cat(E)	(Uses *Item* subclasses)
AbItem(C:1att:0ops)	
Holder(G)	
– *LoanColl*(S)	
– *OtherHolder*(S)	
– *User*(S)	*Borrower*(C:3att:3ops)
	– *UGBorrower*(as above)
	– *PGBorrower*(as above)
	– *ACBorrower*(as above)
	Librarian(C:0att:3ops)
(Uses ℕ)	*Money*(G)
Status(E)	(Uses *Borrower* subclasses)
Clock(C:5att:1op)	
Date(ℕ)	*Date*(G)
Time(Fin. subs. of ℕ)	*Time*(G)

G = given type

S = subset of given type
E = enumerated type
C = class type (with nos. of attrs. and visible ops.)
ℕ = natural numbers

Some notable differences between the specifications are as follows:

- *Library stock.* The EVS stock is unchanging and undivided. The IVS stock is divided between a store, main and short loan collections, and a reference collection; and both the stock and the allocation of items within it are allowed to change. This is an arbitrary modelling decision, and is not due to the ODP viewpoints framework.

- *Roles.* The EVS identifies roles corresponding to the types of borrower, and encapsulates their state and operations by defining Object-Z classes for each type of borrower. The roles are thereby fixed, so that a borrower cannot leave or join the library, or change membership type. By contrast, the IVS effectively allows the role of an object to change (although roles are not identified as fundamental concepts in the Information Viewpoint), since it allows the status of library users to change. The behaviour associated with each user status is specified by global functions, rather than being encapsulated in classes.

- *Range of library operations.* At the library level, the EVS defines three visible operations: issuing an item; returning an item; and paying off all or part of a fine. The IVS defines an extra 20, including operations for changing and reorganising the stock, and for querying aspects of the library state. By their nature, Enterprise Viewpoint specifications are likely to include only a fraction of the operations found in an Information Viewpoint specification.

- *Loan information.* The IVS assigns each loan a unique number, in order of issuance, as well as an issue time and date, and a due date. It also records the collection that the loan is from. When a loan is returned, its return time and date are recorded in the library state. By contrast, the EVS does not record issue time, source (because different collections are not distinguished), return time, or return date, and does not assign unique identifiers to loan objects. This illustrates the general point that one viewpoint might record more detail than another.

- *Clocks, dates, and times.* The IVS explicitly formalises clocks, dates, and times. The EVS implicitly assumes some non-standard semantic features, such as special 'built-in' date and time variables, and events which occur as soon as their preconditions become true. Building some general templates or modules to handle temporal concepts would be useful.

- *Entities not represented.* The EVS represents librarians, whereas the IVS does not. Conversely, the IVS represents other entities which the EVS does not, such as clocks, library collections, abstract items, types of user status, and categories of item (although the EVS represents the latter two aspects using subclasses of *Borrower* and *Item*). This is a specific instance of a general problem with partial specification.

- *Interaction with component objects.* Both specifications assume that state attributes of component objects (e.g. of items) can be accessed by the top-level (library) class and used in state invariants — a practice not permitted by a fully abstract semantics (see [7]), which only allows component objects to be manipulated via their operations. The IVS does, however, adhere to the principle that a component object's state attributes cannot change without an operation on that object; whereas the EVS appears to allow such attributes to change via an 'internal' operation, or to preserve an invariant, without a visible external operation on the object. (The invariant in question is in the EVS library class, and it constrains the 'loans' attribute of all the items belonging to the stock.)

4.2. Discussion of differences

Some differences between the two specifications reflect arbitrary decisions about what to formalise or how to formalise it. Given that an object-oriented formalism is being used, one of the dimensions of variation is the choice of object classes, and the distribution of attributes and operations between them. The method used to derive the EVS identifies classes corresponding to types of passive objects (which have no operations), and classes corresponding to 'roles' (types of active objects, which do have operations). In the library case, this leads to a specification with a lot of attributes and operations at the level of component objects. Since the library itself is not classified as a role, the three operations at the library level are then defined as synchronised executions of component-level operations, e.g. issuing a book involves synchronising a borrower's *Borrow* operation, a librarian's *Issue* operation, and an item's *CheckOut* operation.

By contrast, in the IVS, the only component operations are the *Clock* class's *Tick* operation, and the *Item* class's *MoveItem* and *ChangeItemHome* operations. Several component object types (e.g. the *Date*, *Time*, *User*, *Holder*, *LoanColl*, and *OtherHolder* types) have no state attributes or operations.

Overall, the IVS is more detailed, and records cumulative information which the EVS does not — e.g. the EVS library state does not retain completed loan objects, or their actual return dates and times.

The method used to generate the EVS identifies Enterprise Viewpoint 'roles' with Object-Z classes. This encapsulates information about each role, but has the disadvantage that roles are thereby associated with fixed classes of objects, so that an object cannot change its role. For this reason, the EVS cannot represent changes to a person's library membership status.

5. Unification and refinement

5.1. Conventional refinement in Z

In *data refinement* in Z (see [12]), an abstract ADT is refined to a more concrete one by changing the state space and operations. Let the abstract and concrete ADT's be $(AStates, AInit, \{AOp_i \mid i \in I\})$ and $(CStates, CInit, \{COp_i \mid i \in I\})$, respectively (where *AInit* and *CInit* are initialisation predicates). The concrete ADT refines the abstract one, with respect to a *retrieval relation Retr* between their state spaces, if and only if the following conditions hold:

- *Initialisation*. Every concrete initial state in *CInit* is related by *Retr* to some abstract initial state in *AInit*.

- *Inputs and outputs*. The operations AOp_i and COp_i have the same input and output variables (if any).

- *Applicability*. If a given abstract pre-state *AState* and set of inputs satisfies the pre-condition of AOp_i, then any concrete pre-state *CState* related to *AState* by *Retr* satisfies the pre-condition of COp_i, given the same inputs.

- *Correctness*. For any pre-states *AState* and *CState* related by *Retr*, if *AState* satisfies the pre-condition of AOp_i, and $CState'$ is a concrete post-state reachable from *CState* via COp_i, then there is some abstract post-state $AState'$ related by *Retr* to $CState'$.

The conditions above ensure that the concrete ADT 'simulates' the abstract ADT, in the sense that every behaviour of the concrete ADT corresponds, via the retrieval relation, to a behaviour of the abstract ADT. Two assumptions made here are that: (a) the two ADTs have matching indexed sets of operations; and (b) matching operations have the same input and output variables. For partial specification in Object-Z, these assumptions seem too inflexible.

5.2. Approaches to unification

This section considers how the two viewpoint specifications of the library system might be refined into a single specification. One general strategy might involve rules for 'flattening' each specification, replacing component object classes by additional, appropriately typed higher-level attributes, until each specification is reduced to a single top-level class. Data and operation refinement techniques similar to those conventionally used in standard Z could then be applied to find a common refinement. An alternative strategy might be to define rules for developing a merged specification which retains all the classes of the two specifications, but establishes identities between some classes, refines individual classes, and adds invariants to relate various classes. In practice, a combination of the two approaches could be used — the guiding consideration being the pragmatic one of choosing whichever approach minimises the knock-on effects on the unification process as a whole.

In Z, refinement is applied to a specification representing a single ADT. By contrast, a typical Object-Z specification represents a set of interrelated ADTs — one for each class — although one ADT is identified with the overall system being modelled. This suggests that some refinement rules for Object-Z should operate at a higher level than data refinement rules in Z, i.e. they should relate two finite sets of ADTs, rather than two individual ADTs. Moreover, it should be possible to 'promote' certain refinements on individual ADTs to refinements of the whole specification.

In this section, an indexed collection of ADTs corresponding to an Object-Z specification will be written as a tuple $(\{ADT_j \mid j \in J\}, j_0, \mathit{Inherits}, \mathit{CompOf})$, where:

- j_0 is the index of the 'system' ADT ($j_0 \in J$).
- *Inherits* is the full inheritance relation on the set of ADTs, i.e. the transitive closure of the 'child' (immediate subclass) relation.
- *CompOf* is the 'component of' relation on the set of ADTs, i.e. the transitive closure of the 'immediate component' relation, where type Y is an immediate component of type X if and only if X is a class with a state attribute of type Y or of a higher-order type involving Y and/or other types (e.g. $\mathbb{P}\, Y$, $Y \times Z$).
- ADT_{j_0} is the least element of *CompOf*, so that every other type is a component type (directly or indirectly) of ADT_{j_0}.

In both library specifications, *Inherits* defines a tree, not a DAG, so there is no multiple inheritance. However, *CompOf* is a partial order in the IVS, but not in the EVS, in which some classes are mutual components.

Each ADT is a tuple $(\mathit{AStates}, \mathit{AInit}, \{AOp_i \mid i \in I\})$. In so-called *conformal* refinement steps, the index set I does not change, and COp_i must have the same input and output variables as AOp_i. Some of the rules now proposed, however, do not abide by these restrictions, as discussed in the next section.

5.3. Types of development step

This section looks at the kinds of 'refinement' steps made in the library unification, and attempts to isolate some more general principles. The discussion is informal, but the aim is to identify aspects worthy of more detailed, formal investigation.

Conventional ADT refinement. Conventional ADT data refinement of a class should be possible, via the definition of a retrieval relation between two disjoint state spaces, and the specification of new operations corresponding to the old ones. Weakening of operation pre-conditions should not be allowed, because the 'firing' interpretation of pre-conditions is assumed in Object-Z. Also, in the context of a collection of ADTs, refining one will in general have implications for others, as previously noted — for example, if the operational interface of one class changes, other classes may require modification. One factor that has to be considered when applying conventional refinement rules to an ADT is how to define the post-state, given operations of the form $x.Op$ on object-valued attributes such as x, which change the state of the object to which x refers, without changing x itself, which denotes a reference or pointer to the object.

Adding state attributes. Most unifications of corresponding classes in the library example involve adding attributes to the original state schemas, and possibly relating them to the original attributes by means of new invariants. This seems unproblematical in most cases (although possibly not when the new attributes are of types which themselves refer back, directly or indirectly, to the class type to which the attributes are being added — as discussed shortly). Adding new attributes may require the Δ-lists of existing operations to be modified, unless the new attributes are secondary.

Adding operations. The library case study also raises the issue of extending the possible behaviour by adding operations. For example, the EVS assumes fixed sets of library items and library members, whereas the IVS has operations that can change those sets. One view of this is that such differences make two specifications in-consistent. Another view — natural in the context of viewpoints — is that the initial specifications should be thought of as partial — in which case it ought to be permissible to add operations, the result being a more complete description of the possible behaviour. In terms of an ADT with operations indexed by a set I, this means that I is extended to a proper superset I', without affecting the indexing of existing operations.

A step of this kind is most likely (as in the library case) to be applied to the top-level 'system' class, in which case no further changes are implied. If, however, operations are added to a component class, the new operations will be redundant, unless used to define new higher-level operations.

Adding ADTs. The library case involves adding classes, given types, etc., to each specification, in an attempt to find a unified specification. Any ADT added needs to be meet any constraints assumed regarding the *Inherits* and *CompOf* relations, e.g. that they should be hierarchical.

Replacing many operations by one. In the library case study, two EVS operations without inputs (*CheckIn* and *CheckOut*) were found to correspond approximately to one IVS operation (*MoveItem*) with two input variables. This suggests a kind of development step in which two or more operations on a state schema are replaced by one, with an additional inputs or inputs in the single operation providing a 'menu' of options, allowing it to have the same effect on state variables as any of the individual operations which it replaces. Alternatively, the number of inputs might be the same, but the number of possible value assignments to the inputs might be increased, as e.g. when the type of a single input variable is changed from $\mathbb{B}$ (Boolean) to $\mathbb{N}$ (natural number).

Conventionally, an ADT is a tuple $(AStates, AInit, \{AOp_i \mid i \in I\})$, involving a one-to-one function from an indexing set I to the set of operations for the state space *AStates*. If several operations can be refined into one, this suggests that the indexing function should not have to be one-to-one, or in other words, that an indexed bag of operations should be used, rather than an indexed set. After an operation reduction step, the indexing function should have the same value — i.e. the new, single operation — for several index values, each formerly the index for one of the multiple operations replaced.

Global declarations. In the library case study, some information — e.g. borrowing rights — specified by component classes in one specification, is specified by global functions in the other. This seems justified when the different attribute values of the subclasses involved are fixed anyway (which is true of the *allowance* attribute of the EVS *Borrower* subclasses).

Globally declared functions and constants behave like fixed-value attributes of every class. Development towards a unified specification may involve imposing further constraints on globally declared quantities. For example, in the library case, specific values are given to some globally defined quantities from the IVS, so that they conform to the values of corresponding quantities specified in the EVS.

Removing classes. The library unification involves removing the inherited EVS subclasses *Book* and *Periodical* of the class type *Item*. This is justified by the presence in the unified *Item* schema of the extra attribute $cat : Cat$, which classifies items into books and periodicals. In more abstract terms, what is involved here is a class $X0$, with subclasses $X1$, ..., Xn, being used in the context of a higher-level class in which there is an attribute $xs : \mathbb{F} \downarrow X$ and an invariant of the form (where $\veebar$ denotes exclusive OR):

$$\forall x : xs \bullet x \in X1 \veebar ... \veebar x \in Xn$$

The library case study also involves the replacement of the EVS *Borrower* class hierarchy by a subset (*User*) of a given type (*Holder*), which is justified on the grounds that it implies fewer changes during the overall unification than would otherwise be the case. Information held in the original class hierarchy, concerning the borrowing rights of different kinds of borrower, is specified alternatively by means of global functions.

Developing attribute types. Another way to 'develop' Object-Z classes is to develop the types of state attributes. For example, when the top-level IVS library schema is developed into the unified library schema, the values of the attribute *loans_out* are sets of the unified, refined *Loan* class objects. Thus the development of a class X often constitutes an implicit development of other classes with attributes of type X, or of compound types involving X (e.g. $\mathbb{P} X$, $X \to Y$, etc.). Moreover, the development of classes which explicitly refer to X may have further knock-on effects on other classes that do not directly refer to X. Thus the development of one class cannot be considered in isolation from that of others.

In hierarchical Object-Z specifications, the knock-on effects of developing a type will flow only upwards, towards the top-level class. In specifications with mutually inter-defined types, the flow will be partly circular. For example, class type X may have an attribute of type Y, and vice versa. Developing X will then amount to developing Y, but this in turn will be a further development of X, and so on. In other cases, X and Y may be linked indirectly via other classes. Whether the circularity is direct or indirect, it makes the interpretation of development steps more difficult, and may in some cases be vicious. This is an area worth further investigation regarding the semantics of Object-Z specifications, and the nature of refinement in Object-Z.

6. Conclusion

This paper has used a case study of a lending library to compare two Object-Z specifications representing the ODP Enterprise and Information Viewpoints. The Information Viewpoint Specification (IVS) is more detailed than the Enterprise Viewpoint Specification (EVS), and represents cumulative information about loans which the EVS does not. It also represents a greater range of behaviour, allowing items to be moved between collections or into and out of stock, and permitting users to join, leave, or change membership type. Other significant differences are a greater use of component object attributes and operations in the EVS, and a more explicit model of time in the IVS.

Attempting to unify the two library specifications raises many issues of wider interest than those concerning only ODP viewpoints, relating to the refinement steps which should be allowed for Object-Z specifications interpreted as partial specifications. Conventional Z refinement is not adequate for this task, and needs to be extended in several ways.

The types of development step worthy of more formal investigation include: adding new operations; removing classes (partial 'flattening' of a specification); replacing several operations by one, which has more inputs; and developing the types of class attributes. Regarding the last of these, providing a satisfactory account of Object-Z refinement may be harder when a specification has mutually inter-defined classes.

7. References

[1] E. Boiten, J. Derrick, H. Bowman, and M. Steen, "Constructive Consistency Checking for Partial Specification in Z", Science of Computer Programming, 35, 1999, pp. 29–75.

[2] E. Boiten and J. Derrick, "Liberating Data Refinement", Mathematics of Program Construction, eds. J.N. Oliveira and R.C. Backhouse, LNCS, Springer, 2000 (to appear).

[3] H. Bowman, E. Boiten, J. Derrick, and M. Steen, "Viewpoint Consistency in ODP, a General Interpretation", Formal Methods for Open Object-Based Distributed Systems, eds. E. Najm and J.-B. Stefani, Chapman & Hall, March 1996, pp. 189–204.

[4] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development", Int. Jour. on Software Engineering and Knowledge Engineering, 1992, 2(1), pp. 31–58.

[5] He Jifeng, C.A.R. Hoare, and J.W. Sanders, "Data Refinement Refined", Proc. ESOP 86, eds. B. Robinet and R. Wilhelm, LNCS, Springer, vol. 213, 1986, pp. 187–196.

[6] P.F. Linington, "RM-ODP: The Architecture", ICODP, eds. K. Raymond and L. Armstrong, Chapman and Hall, February 1995, Brisbane, Australia, pp. 15–33

[7] G. Smith, "A Fully Abstract Semantics of Classes for Object-Z", Formal Aspects of Computing, 7(3), 1995, pp. 289–313.

[8] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 2000.

[9] J.M. Spivey, The Z Notation: A Reference Manual, Prentice Hall, 2nd edition, 1992.

[10] M. Steen and J. Derrick, "Formalising ODP Enterprise Policies", EDOC, 1999, University of Mannheim, Germany, IEEE Publishing.

[11] C. Taylor, "Comparison of ODP Viewpoint Specifications in Object-Z: A Case Study", University of Kent, Tech. Rep. No. 7-00, March 2000.

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