An Object-Oriented Structuring for Z based on Views

Nuno Amálio, Fiona Polack, and Susan Stepney

Department of Computer Science, University of York, York, YO10 5DD, UK
{namalio,fiona,susan}@cs.york.ac.uk

Abstract. There is significant interest in the use of Z in conjunction with object-orientation. Here we present a new approach to structuring Z specifications in an object-oriented (OO) style. Our structuring is based on views, it uses the schema calculus, and it does not extend Z. The resulting OO Z specifications are comprehensible, modular, and conceptually clear. The modularity of the new approach supports a template-instantiation approach to expressing OO models in Z; practical formal verification and validation of the model can be undertaken using meta-proof, meta-lemmas, and formal snapshots.
Keywords: Z, object-orientation.

1 Introduction

For more than a decade, there has been interest in structuring Z specifications in an object-oriented (OO) style [1]. Researchers quickly realised that Z does not directly support object-orientation; fundamental OO concepts such as object and class are not Z language primitives. This has resulted in extensions, such as Object-Z [2], designed to facilitate the structuring of Z-based specifications in an OO style. However, this more natural way of expressing OO properties comes at a cost: a more complex language semantics, and reduced flexibility. This has implications in the language's proof and refinement theories, which also become more complex.

Z is a simple language based on typed set theory and first order logic with a simple structuring mechanism, the schema. It has a mathematical, rather than computational, semantics. This makes it flexible and extensible, allowing structuring based on different computational models.

Here we present a new approach to structuring Z specifications in an OO style, without extending the Z language. The approach has emerged in the context of developing a semantic model to represent abstract UML models [3–5], and the example used to illustrate the approach here is based on a simple UML class model. The full approach is not restricted to class models, to UML, or to any particular variant of OO semantics.

Our approach builds on existing research, reviewed in [6]. It has some novel features that enhance the comprehensibility, abstraction and modularity of the Z model:

– It separates concerns effectively by being based on views, following a views structuring approach for Z [7].
– It is very modular, which is achieved by being based on the schema calculus. This allows us to represent concepts as modules (schemas), that can be composed with other modules to form the whole.

2 The Structuring

This section introduces our structuring, explaining how the components of OO are represented and the views structure adopted, we also introduce our domain toolkit of templates, which allows the generation of models following this structuring by instantiating templates. In the following section, an example model is built up following this structuring.

2.1 Objects, Classes, Associations and Systems

OO models are structured around the concept of the *object*. Objects are characterised by an identity (distinguishing one object from all others) and observable properties. The building blocks of our models are object-based structures, namely *class*, which represents a set of objects; *association*, which represents relations between classes of objects; and *system*, which composes classes and their associations and represents global scope properties.

An OO class has a dual meaning. Class *intension* defines a class in terms of the properties shared by the objects of that class (for example, a class *Person* with properties *name* and *address*). Class *extension* defines a class in terms of the currently existing object instances of that class (for example, *Person* is $\{MrSmith, MrAnderson, MsFitzgerald\}$). This duality, inspired by the definitions of a set in set theory, is reflected in the way we represent classes: each class has one intensional and one extensional representation.

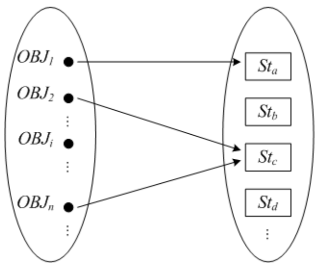

Fig. 1. The set of all object atoms OBJ, the set of all object states St (the class intension), and the mapping from existing objects to their current states (the class extension).

In our approach, Figure 1, objects are atoms (individuals represented in Z as elements of a given set). The class intension defines the set of all possible object states. A function maps the existing object atoms to their current states (the class extension). Representing objects as atoms ensures the identity property necessary for objects. Separation of concerns is achieved, because objects, class intensions, and class extensions are all related, but separately represented.

Associations express relationships between classes. An association denotes a set of object tuples, where each tuple describes the objects being related (or linked). In our approach, associations are represented as a Z relation between objects.

Systems are used to assemble the local structures, classes, and associations into more global ones; systems also include invariants (or constraints) whose scope goes beyond local structures.

The representation of class intensions and extensions, associations and systems follows a Z state and operations style defined through the schema calculus. Each is represented as a Z abstract data type (ADT), comprising a state, initialisation, operations, and finalisation.

2.2 Views

A view [7] is a partial specification of a program consisting of a state space and a set of operations. A full specification is obtained by composing several views, linking them through their states and through their operations.

We use views because not all properties of OO models fit into a single representation. For example, we cannot capture the three representations of classes — atoms, intension, extension — in a single view. Also, views have proved to be an effective means of achieving clear separation of concerns in the specification. This conceptual clarity helps not only in writing and reading the Z models, but also in formal verification and validation. The clear separation of views allows, among other things, a simple solution to the frame problem in system operations (below).

The views that we use are closely related to our basic OO structures. The *structural* view defines sets of object atoms, and captures properties of class structures from an atom perspective of classes. The *intensional* and *extensional* views represent, respectively, the intensional and extensional meanings of classes. The *relational* view represents associations. The *global* view represents systems. The structural view is the only one that does not follow a Z state and operations style; it is a conceptual view, enhancing the conceptual cohesion of the whole structuring. The other views are collections of ADTs, representing class intensions, class extensions, associations, and systems.

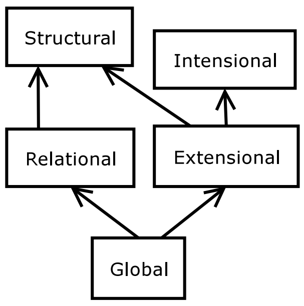

Fig. 2. The views of our structuring and dependency relationships (arrow means dependency).

Figure 2 shows the dependencies among these views. The structural view introduces global names that allow the relational and extensional views to be built independently whilst sharing the same vocabulary, and then to be linked in the global view. The intensional view defines the possible states of objects; these definitions are used in the extensional view to define the mapping between existing objects (atoms) and their current state.

We use different mechanisms to link views. The structural view defines global names, which are then used (directly or as a Z *parent* section) in the descriptions of the extensional, relational and global views. The extensional and intensional views are linked using Z promotion [8, 9]: a class extension includes a collection of state intensions (as a mapping from object atom to state), the intensional operations are then promoted to be applicable to all the objects of the class. Finally, the link between the description of the global view and the ones from the relational and extensional views is established through Z schema conjunction.

2.3 A domain toolkit of templates

The use of views, and the resultant highly modular Z model, allows a systematic approach to constructing the model. We use *templates* to capture the structure of the Z model. Templates are pieces of instantiable Z, which we use to parameterise all sorts of Z phrases, such as axiomatic definitions, schemas and instantiations of Z generics. Template descriptions represent meta-level concepts, such as, class intension, class extension, association and global constraint. Templates are instantiated by reference to the labels of the equivalent concepts in conventional OO diagrams [3, 5] (e.g. the state definition of a class intension is obtained by instantiating the proper template with reference to the class name, and names of composing attributes), and by other model information not expressible in terms of diagrams (e.g. a state invariant of a class intension, which is not represented diagrammatically).

One particular interpretation of a modelling concept is represented by a set of templates. Alternative interpretations of a concept (e.g. association) can be defined by providing another set of templates. This allows us to construct Z models based on variant OO interpretations by selecting an appropriate alternative set of templates.

Standard conjectures, such as an initialisation conjecture, can also be expressed in template form. In our work, various template conjectures have been subject to formal analysis, leading to a number of meta-theorems and meta-lemmas. For correctly-instantiated specification templates, these reduce the proof of standard conjectures to a trivial exercise [10].

Our templates and associated meta-theorems constitute our domain toolkit for building OO Z specifications. Like the Z mathematical toolkit, they provide generic definitions and laws to construct and reason about Z specifications. Unlike the Z mathematical toolkit, which is based on Z generics, our toolkit is based, essentially, on templates; the use of templates allows us to increase the level of scope of generic definitions: we can generalise whole specifications and be closer the application domain. The Z model given below has been constructed and consistency-checked by using templates and meta-theorems from the domain toolkit. This strategy of building toolkits of generic definitions is based on the pattern application-oriented theory [11].

3 Example

We illustrate our OO structuring with the specification of a trivial bank system. The static structure is captured by a UML class diagram, figure 3.

- *Customer* represents the bank's customers; the attributes record the *name* of a customer, its *address*, and *type* (either *company* or *personal*).
- *Account* represents the accounts managed by the bank; the attributes record the account number (*accountNo*), the *balance*, and the *type* of account (either *current* or *savings*).
- The association *Holds* relates *Customer*s and their *Account*s; a *Customer* may have zero or more accounts; an *Account* must have one customer.

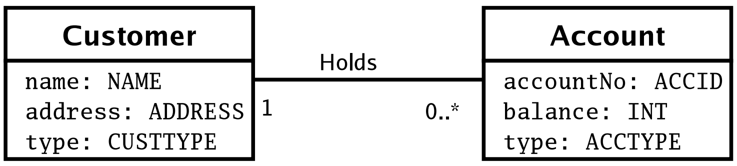

Fig. 3. The example UML model. See text for various Z annotations.

The trivial bank system has the following constraints:

1. Savings accounts cannot have negative balances.
2. The total balance of all the bank's accounts must not be negative.
3. Customers of type *company* cannot hold savings accounts.

The system provides the following operations:

- *Open Account* : open a new account for an existing bank customer.
- *Deposit* : deposit some money into one account
- *Withdraw* : withdraw money from one account
- *Get Balance* : get the balance of one account
- *Get Customer Accounts* : get all the accounts of a certain bank customer
- *Get Accounts in Debt* : get all the accounts that are in debt
- *Delete Account* : delete one account from the system

We start the formal model of the trivial bank system by specifying the state space of the system, and define operations on the state. Only illustrative components are given; like components are specified in similar ways, instantiated from the same templates.

3.1 Specifying Z State

The Z state is constructed from the five views. We use systematic naming conventions, based on [12, chap. 8]. A name starting with the letter $\mathbb{S}$ designates a concept from the extensional view, and a name starting with the letter $\mathbb{A}$ designates a concept from the relational view.

Structural View. Z does not support subtyping; we define a single Z type to represent all object atoms, as this allows us to model OO specialisation hierarchies in a natural manner (not illustrated here).

$$[OBJECT]$$

Each class in the model has its own subset of object atoms. In addition, the object sets for the classes Customer and Account are disjoint, as these classes are not related by a specialisation relation:

$$\begin{array}{|l}
CustomerOs, AccountOs : \mathbb{P}\ OBJECT \\
\hline
\mathrm{disjoint}\langle CustomerOs, AccountOs \rangle
\end{array}$$

Intensional View. The state intension defines classes in terms of attributes, attribute types, and, where required, a class invariant. Attribute types are defined as appropriate. For example, for Account we define $ACCID$ as a given set, and $ACCTYPE$ as a Z free type:

$$[ACCID] \qquad\qquad ACCTYPE ::= current \mid savings$$

The state and initialisation of Account intension is given below. The state $Account$ defines the state attributes, and expresses the first system constraint (*savings accounts cannot have negative balances*). The initialisation makes an assignment of values to class attributes.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Account\ }} \\
accountNo : ACCID \\
balance : \mathbb{N} \\
type : ACCTYPE \\
\hline
type = savings \Rightarrow 0 \leq balance \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ AccountInit\ }} \\
Account\ ' \\
accountNo? : ACCID \\
type? : ACCTYPE \\
\hline
accountNo' = accountNo? \\
balance' = 0 \\
type' = type? \\
\hline
\end{array}$$

Extensional View. A class state extension defines the set of all *existing* objects (a subset of the class' object set), and a function that maps object atoms to their state intensions. The structure is expressed in a generic from our domain toolkit:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathbb{S}\mathrm{Gen}\ [OSET, OSTATE]\ }} \\
objs : \mathbb{P}\ OSET \\
objSt : OSET \nrightarrow OSTATE \\
\hline
\mathrm{dom}\ objSt = objs \\
\hline
\end{array}$$

Actual class state extensions are instantiations of this generic. For example, the Account state extension instantiates the generic and expresses the second system constraints (*the total of all the account balances must not be negative*):

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathbb{S}\mathit{Account}} \\
\hline
\mathbb{S}\mathrm{Gen}[\mathit{AccountOs}, \mathit{Account}][\mathit{accounts}/\mathit{objs}, \mathit{accountSt}/\mathit{objSt}] \\
\hline
0 \leq \Sigma\{\ a : \mathit{accounts} \bullet a \mapsto (\mathit{accountSt}\ a).\mathit{balance}\ \} \\
\hline
\end{array}$$

(See the appendix for the definition of Σ.) The instantiation, guided by a toolkit template, includes the renaming of generic components, to avoid name clashing when component schemas are composed to make the system schema.

The initialisation of the extension assigns both the set of existing objects and the set of object atoms to state mappings to the empty set; in the initial state there are no objects.

$$\mathbb{S}\mathit{AccountInit} == [\ \mathbb{S}\mathit{Account}\ '\ \mid \mathit{accounts}' = \varnothing \land \mathit{accountSt}' = \varnothing\ ]$$

Relational View. In the relational view we define the association state as a Z relation between the object sets of the classes being associated.

The $\mathbb{A}\mathit{Holds}$ state definition defines the relationship between the object sets of the classes Customer and Account. The initialisation of the relationship states that the set of existing links is empty; in the initial state there are no objects, hence, no links between them.

$$\mathbb{A}\mathit{Holds} == [\ \mathit{holds} : \mathit{CustomerOs} \leftrightarrow \mathit{AccountOs}\ ]$$
$$\mathbb{A}\mathit{HoldsInit} == [\ \mathbb{A}\mathit{Holds}\ '\ \mid \mathit{holds}' = \varnothing\ ]$$

At this stage, we do not constrain the *holds* relation to reflect the multiplicity of the association; multiplicities are defined on existing objects, which are defined in the extensional view and are not directly accessible from the relation view. An alternative modelling of associations that included extensions of the associated classes in the association state schema, to allow multiplicity constraints here, would break the separation that exists between relational and extensional views.

Global View. In the global view we compose classes and associations, and express properties of global scope, to form systems. The system state includes classes and association of the system, link invariants between associations and class extensions, and global scope constraints. The system initialisation is the initialisation of the system's components. In the illustration of the trivial bank system, the system has two classes and a single association between them; in general, however, the system is made up of subsystems comprising classes and their linking associations; invariants are added at the appropriate scope, to a subsystem or the full system, as appropriate.

In the global view, each association has a schema expressing the appropriate association link invariant. To maintain the modular structuring, global constraints are expressed in separate schemas, and then added to the predicate of the system schema (based on the Name Predicates pattern [11]).

$\mathit{Link}\mathbb{A}\mathit{Holds}$ expresses the link invariant of the association *Holds*:

$$\begin{array}{|l}
\hline \textit{Link}\mathbb{A}\textit{Holds} \\
\mathbb{S}\textit{Customer};\ \mathbb{S}\textit{Account};\ \mathbb{A}\textit{Holds} \\
\hline \textit{holds} \in \mathrm{Rel}_{1,*}[\textit{customers}, \textit{accounts}] \\
\hline
\end{array}$$

The global view can use both the relational and extensional views, so the schema can include the extensions both of the participating classes and of the association. The predicate constrains the relation representing the association to be the correct multiplicity, using the appropriate association multiplicity generic from the toolkit (see Appendix); it says that the inverse of the relation must be a total function from the set of existing accounts to the set of existing customers. This ensures both the correct association multiplicity $(* \mathinner{\ldotp\ldotp} 1)$, and that the relation refers to existing objects only.

The third constraint of the system (*customers of type company cannot hold savings accounts*) involves concepts from multiple views. It is expressed in the global view as *ConstraintCompanyNoSavings*:

$$\begin{array}{|l}
\hline \textit{ConstraintCompanyNoSavings} \\
\mathbb{S}\textit{Customer};\ \mathbb{S}\textit{Account};\ \mathbb{A}\textit{Holds} \\
\hline \{\ oC : \textit{customers} \mid (\textit{customerSt}\ oC).\textit{type} = \textit{company}\ \} \\
\qquad \lhd\ \textit{holds} \rhd \{\ oA : \textit{accounts} \mid (\textit{accountSt}\ oA).\textit{type} = \textit{savings}\ \} \\
= \varnothing \\
\hline
\end{array}$$

The system schema is defined by conjoining all the class extensions and associations, association link invariants, and global constraints:

$$\begin{array}{|l}
\hline \textit{System} \\
\mathbb{S}\textit{Customer};\ \mathbb{S}\textit{Account};\ \mathbb{A}\textit{Holds} \\
\hline \textit{Link}\mathbb{A}\textit{Holds} \\
\textit{ConstraintCompanyNoSavings} \\
\hline
\end{array}$$

The initialisation of the system consists of initialising the system's components. This is done by conjoining the component initialisations:

$$\textit{SysInit} == \textit{System}\,' \land \mathbb{S}\textit{CustomerInit} \land \mathbb{S}\textit{AccountInit} \land \mathbb{A}\textit{HoldsInit}$$

Note that the after state of the *System* is included in this conjunction, so that the constraints declared globally in *System* are included. This ensures that all the constraints are taken into account in the system initialisation theorem.

3.2 Specifying Z Operations

A system operation in an OO system comprises operations on a number of classes. In modelling system operations in our Z style, we first decompose them into constituent class operations. For example, the operation to open a new account involves the creation of a new *Account* object and the addition of a link, between the new account and its customer, to the association *Holds*. The system operation is the composition of these two operations.

We specify operation components in the intensional, extensional and relational views, and then compose them in the global view to form system operations.

We introduce naming conventions to distinguish update (change state) from observe (do not change state) operations. Following Z conventions, names of update operations include the symbol Δ subscripted, whereas observe ones include the symbol Ξ.

Intensional View. The operations of the intensional view specify state transitions or observations on the state of a single class object.

We distinguish two kinds of operations. *Update operations* change the state of objects; they effect a state transition. *Observe operations* leave the state of objects unchanged, performing queries on the current state of objects. We may also need to specify the *finalisation*, which expresses a condition for objects of a class to cease their existence.

The system operations *Deposit* and *Withdraw* change the state of one account object: the *balance* is incremented or decremented by a certain amount. The intensional view is:

$$\begin{array}{|l}
\multicolumn{1}{l}{Account_{\Delta}\,Withdraw} \\
\Delta Account \\
amount? : \mathbb{N} \\
\hline
accountNo' = accountNo \\
type' = type \\
balance' = balance - amount? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Account_{\Delta}Deposit} \\
\Delta Account \\
amount? : \mathbb{N} \\
\hline
accountNo' = accountNo \\
type' = type \\
balance' = balance + amount? \\
\hline
\end{array}$$

The system operation *GetBalance* observes the state of one *Account* object, specifically the *Account.balance* state attribute:

$$Account_{\Xi}\,GetBalance == [\ \Xi Account;\ balance! : \mathbb{Z} \mid balance! = balance\ ]$$

We want to be able to delete accounts, but only only when their balance is zero. This finalisation condition is specified on *Account* objects:

$$AccountFin == [\ Account \mid balance = 0\ ]$$

Extensional View. This view defines operations that are applicable to all existing objects of a class. Most operations of this view are defined by *promoting* operations from the intensional view.

Z Promotion uses *framing* schemas to promote local operations to a global state. Our approach provides framing schemas customised to our class state extensions, drawing on our work on promotion patterns [9, 11]. There is one framing schema for each kind of operation, the usual *new*, *update*, and *delete* framing schemas; in addition we introduce an *observe* framing schema. For example, the update and observe framing schemas for the class Account are:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Phi\mathbb{S}AccountUpdate} \\
\hline
\Delta\mathbb{S}Account \\
\Delta Account \\
oAccount? : AccountOs \\
\hline
oAccount? \in accounts \\
\theta Account = accountSt\ oAccount? \\
accounts' = accounts \\
accountSt' = accountSt \oplus \\
\quad \{oAccount? \mapsto \theta Account'\} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\Phi\mathbb{S}AccountObserve} \\
\hline
\Xi\mathbb{S}Account \\
\Xi Account \\
oAccount? : AccountOs \\
\hline
oAccount? \in accounts \\
\theta Account = accountSt\ oAccount? \\
\hline
\end{array}
$$

The framing schemas are used to form promoted operations. For example, given the appropriate intensional view operations and framing schemas for the *Account* class, the extensional operations for initialisation, withdrawal, deposit, an account enquiry, and an account deletion are:

$$\mathbb{S}_{\Delta}AccountNew == \exists\ Account\ ' \bullet \Phi\mathbb{S}AccountNew \land AccountInit$$

$$\mathbb{S}_{\Delta}AccountWithdraw == \\ \quad \exists\, \Delta Account \bullet \Phi\mathbb{S}AccountUpdate \land Account_{\Delta}Withdraw$$

$$\mathbb{S}_{\Delta}AccountDeposit == \exists\, \Delta Account \bullet \Phi\mathbb{S}AccountUpdate \land Account_{\Delta}Deposit$$

$$\mathbb{S}_{\Xi}AccountGetBalance == \\ \quad \exists\ \Xi Account \bullet \Phi\mathbb{S}AccountObserve \land Account_{\Xi}GetBalance$$

$$\mathbb{S}_{\Delta}AccountDelete == \exists\, Account \bullet \Phi\mathbb{S}AccountDelete \land AccountFin$$

We may also have operations in the extensional view that are not promotions. For example, the trivial bank may wish to identify all the accounts that are in debt. This is expressed in the extensional view without promotion of an intensional operation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathbb{S}_{\Xi}AccountGetDebtAccounts} \\
\hline
\Xi\mathbb{S}Account \\
osAccount! : \mathbb{P}\ AccountOs \\
\hline
osAccount! = \{\ a : accounts \mid (accountSt\ \ a).balance < 0\ \} \\
\hline
\end{array}
$$

Relational View. Operations in the relational view change or observe the state of associations. Association operations add and remove pairs from the tuples of the association relation, and perform queries on the state of associations.

In the trivial bank system, we need to associate bank customers with new accounts. This is done by adding tuples, consisting of one existing Customer object and one existing Account object, to the Holds association:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathbb{A}_{\Delta} HoldsAdd} \\
\hline
\Delta \mathbb{A} Holds \\
oCustomer? : CustomerOs \\
oAccount? : AccountOs \\
\hline
holds' = holds \cup \{oCustomer? \mapsto oAccount?\} \\
\hline
\end{array}
$$

When a bank account is removed, the link that exists between the account and its customer (association Holds) must also be deleted. The deletion of tuples of the association Holds, given a set of Account objects, is described by the operation $\mathbb{A}_{\Delta} HoldsDelAccount$ (below).

We also want to list all the accounts held by a customer, an observation on the state of *Holds*. The operation $\mathbb{A}_{\Xi} CustomerAccounts$ performs the required observation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathbb{A}_{\Delta} HoldsDelAccount} \\
\hline
\Delta \mathbb{A} Holds \\
osAccount? : \mathbb{P}\, AccountOs \\
\hline
holds' = holds \mathbin{\rhd\!\!\!\!-} osAccount? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\mathbb{A}_{\Xi} CustomerAccounts} \\
\hline
\Xi \mathbb{A} Holds \\
oCustomer? : CustomerOs \\
osAccount! : \mathbb{P}\, AccountOs \\
\hline
osAccount! = holds(\!| \{oCustomer?\} |\!) \\
\hline
\end{array}
$$

Global View. The global view of operations defines system operations that act on the state of the system as a whole. These operations are defined by composition of the operations from the extensional and relational views. This is essentially schema conjunction (except where there is a necessary order of execution of component operations from the separate views). However, as the following example shows, we also need to maintain global constraints, and address the framing problem [13], by making explicit the effect of each operation on the whole system state.

When composing system operations using conjunction, some adjustments may be needed so that the elements of component operations relate correctly across the conjunction. This adjustment involves relating inputs and outputs of component operations.

These issues and their resolutions are explored using the example of a system operation to open a new account. This involves one operation from the extensional view (to create a new account), and one from the relational view (to associate the new account with an existing customer):

$$SysOpenAccount == \mathbb{S}_{\Delta} AccountNew \wedge \mathbb{A}_{\Delta} HoldsAdd$$

First, we need to make an adjustment, because $\mathbb{S}_{\Delta} AccountNew$ outputs an *oAccount*! but the $\mathbb{A}_{\Delta} HoldsAdd$ requires an input *oAccount*?. This is resolved by renaming one (here the input of $\mathbb{A}_{\Delta} HoldsAdd$):

$$SysOpenAccount == \mathbb{S}_{\Delta}AccountNew \wedge \mathbb{A}_{\Delta}HoldsAdd[oAccount!/oAccount?]$$

This simple conjunction does not take into account the global scope constraint of the trivial bank system, that *customers of type company cannot hold savings accounts*, and the violation cannot be determined by formal analysis. To solve this, we need to *lift* component operations to system operations by conjoining $\Delta System$. (This follows the principle of *promotion*, but since promotion in Z refers to a concrete technical mechanism, we term this flavour of promotion *lifting*.)

However, this introduces a frame problem [13]. Component operations specify the change of state of their local components, but a system may include other components whose states should be unchanged by the operation. In this example, the operation should change the states of the Account extension and of Holds, but the extension of Customer, introduced by adding $\Delta System$, should remain unchanged. However, in Z, the state of any component that is not explicitly addressed is undetermined after the operation; any state of $\mathbb{S}Customer$ would satisfy the specification.

In our approach, we define *frames* for system operations. This makes explicit what is to change and what is to remain unchanged. The names of system operation frames are prefixed by Ψ (by analogy to Φ promotion frames), and are formed by conjoining $\Delta System$ with the Ξ (nothing changes) of every system component whose state is to remain unchanged. Thus, the frame for the above example is:

$$\Psi SysAccountHolds == \Delta System \wedge \Xi\mathbb{S}Customer$$

This simple solution is possible because views give the required separation of concerns. Components (classes and associations) are specified independently from each other; when we preceed a component with Ξ we know that we are saying that only this component's state is to remain unchanged and nothing else[1].

The system operation to open an account can now be fully specified by schema conjunction:

$$\begin{aligned} SysOpenAccount == {} & \Psi SysAccountHolds \wedge \mathbb{S}_{\Delta}AccountNew \\ & \wedge \mathbb{A}_{\Delta}HoldsAdd[oAccount!/oAccount?] \end{aligned}$$

In common with the constraints added to the global view of the system state, some operation preconditions can only be expressed in the global view. If, in our example, we want to add a precondition that the new account is associated with an existing bank customer, this is specified in a condition schema:

[1] A version of state extension modelling that included class extensions in the association state schema (for example, to allow the specification of association multiplicity constraints in the relational view) would preclude this simple solution, because $\Xi\mathbb{A}$ would mean that neither the association *nor* the included class extensions could change.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CondIsCustomer}} \\
\mathbb{S}\,\mathit{Customer} \\
\mathit{oCustomer?} : \mathit{CustomerOs} \\
\hline
\mathit{oCustomer?} \in \mathit{customers} \\
\hline
\end{array}$$

and conjoined in the system operation:

$$\begin{aligned}
\mathit{SysOpenAccount} == {} & \Psi \mathit{SysAccountHolds} \land \mathbb{S}_{\Delta}\mathit{AccountNew} \\
& \land \mathbb{A}_{\Delta}\mathit{HoldsAdd}[\mathit{oAccount!}/\mathit{oAccount?}] \land \mathit{CondIsCustomer}
\end{aligned}$$

Other system update operations are similarly defined:

$$\Psi \mathit{AccountOps} == \Delta \mathit{System} \land \Xi \mathbb{S}\,\mathit{Customer} \land \Xi \mathbb{A}\mathit{Holds}$$

$$\mathit{SysWithdraw} == \Psi \mathit{AccountOps} \land \mathbb{S}_{\Delta}\mathit{AccountWithdraw}$$

$$\mathit{SysDeposit} == \Psi \mathit{AccountOps} \land \mathbb{S}_{\Delta}\mathit{AccountDeposit}$$

System observation operations do not require a specific frame; nothing changes in the system, so they can be simply conjoined with $\Xi \mathit{System}$. The system operations *get balance*, *get customer accounts*, and *get accounts in debt* are:

$$\mathit{SysGetBalance} == \Xi \mathit{System} \land \mathbb{S}_{\Xi}\mathit{AccountGetBalance}$$

$$\mathit{SysGetCustAccounts} == \Xi \mathit{System} \land \mathbb{A}_{\Xi}\mathit{CustomerAccounts}$$

$$\mathit{SysGetDebtAccounts} == \Xi \mathit{System} \land \mathbb{S}_{\Xi}\mathit{AccountGetDebtAccounts}$$

The system operation to delete an account is defined in a similar way, but requires a rather more elaborate adjustment to the initial conjunction of the operations to delete the account from the set of existing accounts (operation $\mathbb{S}_{\Delta}\mathit{AccountDelete}$ from the extensional view) and the link to its customer (operation $\mathbb{A}_{\Delta}\mathit{HoldsDelAccount}$ from the relational view). $\mathbb{S}_{\Delta}\mathit{AccountDelete}$ takes as input one account object, however $\mathbb{A}_{\Delta}\mathit{HoldsDelAccount}$ expects a set of objects. The adjustment is made in a connector schema, *ConnAccountOs*, which transforms the single output of $\mathbb{S}_{\Delta}\mathit{AccountDelete}$ to a singleton set:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ConnAccountOs}} \\
\mathit{osAccount?} : \mathbb{P}\,\mathit{AccountOs} \\
\mathit{oAccount?} : \mathit{AccountOs} \\
\hline
\mathit{osAccount?} = \{\mathit{oAccount?}\} \\
\hline
\end{array}$$

The connector is added to the system operation specification to form the correct composition:

$$\begin{aligned}
\mathit{SysDeleteAccount} == {} & \Psi \mathit{SysAccountHolds} \land \mathbb{S}_{\Delta}\mathit{AccountDelete} \\
& \land \mathit{ConnAccountOs} \land \mathbb{A}_{\Delta}\mathit{HoldsDelAccount}
\end{aligned}$$

We could, of course, have avoided need for a connector schema by defining $\mathbb{A}_{\Delta}\mathit{HoldsDelAccount}$ to receive one input object rather then a set. However, the given specification is generated by instantiating our templates; we prefer to keep our operation templates generic and then adjust the connection in the global view; the forms of connection schema can also be provided as templates.

4 Discussion

We have described and illustrated an OO structuring for Z based on views and schema calculus. We are not aware of any other Z-only OO structuring approach that relies entirely on the schema calculus: other approaches all resort to Z axiomatic definitions to express state and operations in one way or another.

A consequence of exploiting the existing structuring mechanisms of Z, the schema and the schema calculus, is that we can systematically compose OO-structured Z specifications. In our approach, we select appropriate templates for each OO component, instantiate these to give named schemas, and then include or conjoin the instantiated components as appropriate to form the Z state and operations. Z schema conjunction is key in our structuring as it allows the propagation of system properties through the composition (a property satisfied by one schema is also satisfied by the conjunction of schemas).

Combined with the schema approach, views provide an effective means to separate concerns, and constituted a powerful conceptual tool. They allowed us to apply the principle of *divide and conquer* to design an aspect-focused structuring, where each individual aspect is conceptually clear and the collection of aspects is cohesive. This effective separation of concerns also allows a simple and elegant solution to the frame problem in the specification of system operations. Because of the basis in Z schemas, we can use Z promotion in the extensional view, to relate the class intension and extension, giving an elegant compositional approach.

The views and the schema approach also allow us to follow a modular approach towards proof. Elsewhere, we show our modular-based approach towards formal model analysis and validation. For example, our approach to consistency-checking [4] takes advantage of the modular structure of our specifications, and is based on the representation of our structures (class intensions and extensions, associations, and systems) as ADTs. To prove the consistency of our specifications, we prove initialisation theorems for the system's components, class intensions and extensions and associations (these are often trivial), and for the whole system; in proving the consistency of the whole system, we use the theorems establishing the consistency of the system's components, reducing the proof to show that the system's global constraints hold in the initial state [10].

To support writing Z models in our OO style, we have devised a domain toolkit of templates [5]. The collection of templates of the toolokit constitutes a meta-level representation of the structuring; the Z specification is created by instantiation of the relevant templates. For example, we can use templates to obtain formal representations, in Z, of UML diagrams; we can provide variant templates for different semantic interpretations of the UML notations. We use our template approach as the mechanism to support the development of *modelling frameworks* [5]: environments to build and analyse models of sequential software systems based on the combined use of UML and Z. In addition to the aspects covered in this paper, our structuring also supports the expression of specialisation (inheritance) hierarchies [5].

It is, of course, possible to write a Z specification following our structuring without instantiating our templates. However, the templates are the basis for *meta-proof* [10]: a set of meta-theorems and meta-lemmas that can be used to reduce the overhead of formal proof when consistency-checking and analysing Z specifications. Having pre-proved key theorems at the meta- or template level, correct instantiation of templates guarantees that the proofs will hold; further proof effort is restricted to local constraints; conversely, if a meta-proof cannot be discharged on an instantiated specification, then the instantiation or the underlying OO (for example, UML) model is wrong.

5 Related Work

Our OO structuring extends that of Hall [14, 15], who introduced the dual representation of classes, intension and extension, in the context of Z. We have introduced views, to achieve greater separation of concerns, and to make the structuring clearer. We have introduced the system structure, and the idea of representing our structures (class intensions and extensions, associations and systems) as ADTs. We have also presented an approach to compose structures: extensions are built from intensions by using promotion, and systems are built by composing class extensions and associations. We have eliminated the need for axiomatic definitions in the definition of operations; extensional class operations are defined by promoting intensional ones.

Our structuring uses the relational interpretation of associations. Alternatively, a representation where associations are interpreted as properties of a class could also be devised [6]. In this setting, a relational view would not be required, and associations would be represented as class attributes in the intensional view. We choose the relation interpretation because we consider it to be more abstract; our aim is to represent abstract UML models.

Utting and Wang propose an OO structuring for Z [16] where state and operations are defined by axiomatic definitions. Objects are atoms, and the relationship between atoms and state fields is given by axiomatically-defined functions, with one function per state field. Operations are also defined axiomatically, as a relation between an object and operation inputs and outputs. One problem with axiomatic-based descriptions in Z is that it is easy to introduce accidental contradictions (especially at the level of complexity of some operations); a contradictory description renders the whole specification unsatisfiable. We find this approach to be cumbersome and difficult to use in practice. We argue that the template support for specification and analysis would not be possible in this approach. Moreover, the resulting specifications are not succinct, and lack modularity; it is not as easy to compose axiomatic definitions as it is to compose schema-based ones.

Our model of objects can be compared to that of the formal specification language Alloy [17], which has a similar semantic basis as Z. Alloy's structuring mechanism, the signature, is inspired by the Z schema. Like schemas, a signature state definition includes the definition of state components (fields). The fundamental difference, however, is that a signature denotes a set of atoms, its fields are also atoms, and signature atoms and field atoms are linked through relations. Unlike Z schemas, this effectively gives identity to signature instances; instances of a Z schema with the same value for its fields denote the same schema object. To overcome this in Z we represent an object atom separately from the object state (a schema), and use a function to map object atoms to their state. In the end, our Z object-model and the Alloy one are not so different. Alloy has one relation between the object atom and each state field; our Z model has a function that relates the object atoms to the entire state schema.

6 Conclusions

We present an OO structuring for Z based on views that relies entirely on the schema calculus to describe both state and operations. The specifications resulting from our structuring are modular, abstract, and comprehensible, following a style that is familiar to and adopted by most Z users.

The OO structuring facilitates traceability (between diagrammatic models and Z models, for example), and supports template-based development and analysis. The approach is the basis for a practical framework for formal development, facilitating verification and validation checks of both formal and informal OO models. The approach is also flexible, as variant OO semantics can be represented simply by selecting the appropriate templates.

Future extensions based on the OO structuring include full support for key UML models (eg. class, state, and interaction diagrams); mutual model refinement, eventually via templates; and tool support for the templates themselves.

Acknowledgements. This research was supported for Amálio by the Portuguese Foundation for Science and Technology under grant 6904/2001.

References

1. Stepney, S., Barden, R., Cooper, D., eds. *Object orientation in Z.* Workshops in Computing. Springer (1992)
2. Smith, G. P. *The Object-Z Specification Language.* Kluwer Academic Publishers (2000)
3. Amálio, N., Stepney, S., Polack, F. Modular UML semantics: Interpretations in Z based on templates and generics. In Van, H. D., Liu, Z., eds., *FACS'03*, 284, pp. 81–100. UNU/IIST Technical Report (2003)
4. Amálio, N., Stepney, S., Polack, F. Formal proof from UML models. In et al, J. D., ed., *ICFEM 2004*, volume 3308 of *LNCS*, pp. 418–433. Springer (2004)
5. Amálio, N., Polack, F., Stepney, S. A modelling and analysis framework for sequential systems I: Modelling. Technical Report YCS-2005, Department of Computer Science, University of York (2005)
6. Amálio, N., Polack, F. Comparison of formalisation approaches of UML class constructs in Z and Object-Z. In Bert et al. [18], pp. 339–358

7. Jackson, D. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389 (1995)
8. Woodcock, J., Davies, J. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall (1996)
9. Stepney, S., Polack, F., Toyn, I. Patterns to guide practical refactoring: examples targetting promotion in Z. In Bert et al. [18], pp. 20–39
10. Amálio, N., Stepney, S., Polack, F. Modular meta-proof for structured specifications (2004). Available at http://www.cs.york.ac.uk/~namalio/publications.html
11. Stepney, S., Polack, F., Toyn, I. A Z patterns catalogue I, specification and refactoring. Technical Report YCS-2003-349, Department of Computer Science, University of York (2003)
12. Barden, R., Stepney, S., Cooper, D. *Z In Practice.* Practitioner Series. Prentice–Hall (1994)
13. Borgida, A., Mylopoulos, J., Reiter, R. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798 (1995)
14. Hall, A. Using Z as a specification calculus for object-oriented systems. In Hoare, C. A. R., Bjørner, D., Langmaack, H., eds., *VDM '90*, volume 428 of *LNCS*, pp. 290–318. Springer (1990)
15. Hall, A. Specifying and interpreting class hierarchies in Z. In Bowen, J., Hall, A., eds., *Z User Workshop, Cambridge*, Workshops in Computing, pp. 120–138. Springer (1994)
16. Utting, M., Wang, S. Object orientation without extending Z. In Bert et al. [18], pp. 319–338
17. Jackson, D., Shlyakhter, I., Sridharan, M. A micromodularity mechanism. In *ACM SIGSOFT Foundation of Software Engineering/ Europoean Software Engineering Conference* (2001)
18. Bert, D., et al., eds. *ZB 2003, Turku, Finland*, volume 2651 of *LNCS*. Springer (2003)

A OO Template Toolkit (excerpt)

Selected Association Multiplicity Generics

$$\mathrm{Rel}_{*,1}[X, Y] == X \rightarrow Y$$
$$\mathrm{Rel}_{1,*}[X, Y] == \{ r : X \leftrightarrow Y \mid r^{\sim} \in \mathrm{Rel}_{*,1}[Y, X] \}$$

Sum over a finite labelled set

$$[L]$$
$$\Sigma : (L \twoheadrightarrow\!\!\!\!\!\shortmid\;\, \mathbb{Z}) \rightarrow \mathbb{Z}$$
$$\Sigma\ \varnothing = 0$$
$$\forall\, l : L;\ n : \mathbb{Z};\ f : L \twoheadrightarrow\!\!\!\!\!\shortmid\;\, \mathbb{Z} \mid l \notin \operatorname{dom} f \bullet \Sigma(\{l \mapsto n\} \cup f) = n + \Sigma\, f$$