Systems Development using Z Generics

Fiona Polack[1] and Susan Stepney[2]

[1] Department of Computer Science, University of York, UK.
fiona@cs.york.ac.uk, (tel +44 1904 432722)
[2] Logica UK Ltd, Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK.
stepneys@logica.com

1 Introduction

In this paper we present a method for using generic components in formal specifications. This approach results in a flexible generic system description that separates the concerns of structure and data types. The generic specification can be extended and modified in a natural manner, to track requirements as they inevitably evolve during the development process. In addition, the specification can readily be specialised to use more concrete data types without the need for a formal refinement, using explicit generic instantiation. Such generic instantiation also allows operation preconditions to be *strengthened*; this is not allowed by classic refinement, but it permits a separation of concerns by allowing preconditions relevant to specialised data types to be added only when they become relevant.

Here we use the Z specification language and a simple entity-relationship form as demonstration notations. No new notation or theory is presented; rather it is the use of Z's generic schemas to structure and specialise a specification that is somewhat different from the classical Z specification style described in much of the literature. We believe that this approach could also be applicable to other formal methods.

2 The Case for Z, and Z Generics

Z is similar to many model based notations: it applies typed set theory and predicate logic to system description; it permits rigorous analysis and proof of system properties; it expresses a system description in precise terms.

Z's most recognisable feature is the *schema box*, used to help structure specifications by grouping together definitions. Z also allows *generic definitions* that may be instantiated on use with any set of any type; schemas may themselves be generic. Most Z specifiers confine their use of generics to global toolkit-style constants used with implicit instantiation; few fully exploit the possibility of generic schemas with explicit instantiation.

2.1 Generic definitions in defining Z toolkits

Z has used generic definitions from its earliest public appearances. In his definitive Z Reference Manual [1] Spivey gives a generic *Mathematical Toolkit* for the

Z language. All Z text books (for example [2–4]) demonstrate generic definitions of operations on sets, sequences and bags, as well as more customised operations.

An example of such a generic definition (used in our case study below) is a generic optionality definition [5].

$$optional[X] == \{ a : \mathbb{F}\, X \mid \#a \leq 1 \}$$

This can be used to model an item that may be present or absent. If y is declared to be $y : optional[Y]$, then y is a set that is either empty (absent) or a singleton (containing a single element of the actual parameter set Y). The type of the set is controlled by the type of the parameter Y.

It may be tempting to see this basic use of Z generic definitions as fulfilling the dream of 1980s programmers, regarding generic data types. Although fundamental to the use of the Z notation, however, these generic definitions do not contribute to the process of developing systems.

2.2 Generics for secondary toolkits

The widespread development of formal/structured integrations in the 1990s (see summary papers [6, 7]) has seen the use of Z generic schemas to define representations of structured model components. For example, the SAZ Method [8–10] includes generic representations of entities and the various forms of relationship encountered in its data modelling notation.

SAZ is an *interpretive* method (guidelines are used to convert a structured specification in to a formal notation), rather than a formal *translation* approach; the SAZ generics form a toolkit that is imported in to the formal document environment, giving a uniform appearance to the formal description.

SAZ generic toolkit definitions have advantages and disadvantages.

On the plus side, they reduce the tedium of transliteration, and provide recognisable components for a reviewer of the formal description. There is little doubt, for example, that the component

$$CustomerSet == EntitySet[CID, Customer]$$

represents a set of entity instances. In addition, such generics conceal or defer low-level decisions about details of components, such as an implementation type for *Customer*.

On the other hand, the component generic definitions become extremely complex, so the simplicity gained in the system description merely masks, rather than removes, a complexity of meaning. For example, the various SAZ relationship generics have formal parameters that are instantiated implicitly in both the declaration and predicate parts. Current Z tool support often fails to expand these generics fully, and then cannot assist in proof of many properties of systems that use them.

Again, these generic usages simply help to express systems; they do not offer any real contribution to the *system development process*.

2.3 Generic systems

The approach advocated here is to use generic definition at the *system level*. This approach has some well-known precursors, although neither has the simplicity or applicability of the approach demonstrated below.

One example is Flinn and Sørensen's CAVIAR case study [11, chapter 5]. This includes generic modules, providing a super-structure for the specification, and promotes the development of generic specification libraries, as the authors advocate. The module concept is very similar to the use below, but uses an extension to the Z notation to bring this about.

Another example is a proof of compliance to a security policy model, summarised in [2, chapter 4]. Here, a generic top level specification is defined, and it is demonstrated that this has the generic security properties. The specification is instantiated with suitable parameters (input structures, output structures, states), without need to re-prove the high level properties. Again, the approach is closely related to that demonstrated below. The entire specification is cast as an axiomatic state transition relation, however.

3 System Development in Z

Literature on system development in Z concentrates on *formal refinement* [23, 2, 4], neglecting simpler or more intuitive development options. Formal refinement in Z as part of the development of commercial applications is a reality (for example, [12, 13]), but it requires considerable effort, and can still justify its costs only for critical applications. Limitations of refinement (in addition to the difficulty of performing them) are also being recognised among the academic formal methods community [14, 15].

Here we explore the use of generic instantiation rather than refinement as the formal process for developing a system. The development process can be seen as a progressive reduction in options, as the detail in the system description increases and becomes more targeted to specific implementation media [16, 17]. We use a simple development lifecycle to illustrate possible development paths. Although this is not taken forward to implementation, the techniques demonstrated can be reiterated until the required level of specialisation is attained.

4 Entity-Relationship description of the Case Study

To illustrate our approach, we use a simple case study derived from the SAZ project [9, 10].

In this section we give an informal description of the system. In the next section we formalise the system state and some sample operations as a generic Z specification. In section 6 we perform a first specialisation, by defining the objects in the data-dictionary, and instantiating the specification with them. In section 7 we describe how the generic structure of the specification makes it relatively easy to extend and modify both the structure and the components seperately.

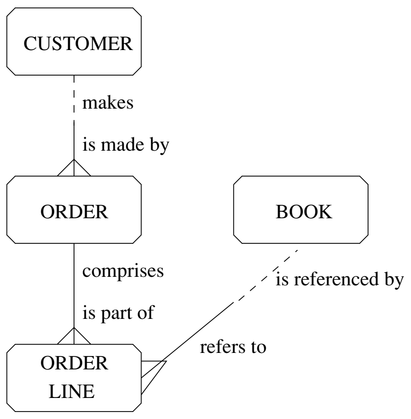

Fig. 1. The data model for the case study, using SSADM v4 notation [18].

4.1 Scenario

A publishing company accepts orders for books. An order must be placed by a customer; details of customers placing orders are kept on file. The company processes orders and notes whether each order line is met in full.

4.2 Data Model

A data structure to support this system is given in figure 1, as an Entity-Relationship Diagram (ERD).

4.3 Events and Processing

The system receives events to create, modify and delete elements of the data structure. These could be modelled dynamically, using entity life histories or state charts; here they are simply described.

Receiving an order: Receipt of an order from a new customer triggers the creation of a new customer instance. Receipt of any order triggers the creation of an order instance, and of a set of instances of order line. The processing includes validation checks to determine whether the customer is known or new, and on the composition of order lines.

Customer and Order modifications: Customer details, especially addresses, may be changed at any time. Order details are never changed; a re-order would be a new instance. However, additional lines might be added to an existing order. Customers are deleted from the records if they have not placed an order for 5 years. Orders are deleted from the records when the customer is deleted, or after 10 years if this is sooner.

Processing an order and its component order lines: An order instance is processed by first processing each of its order lines. The details of the processing are not included here, but involve checking that at least some of the order lines can be met and that an order line is not for the same book as another order line on the same order.

If an order has no order lines that can be met (that is, all supplied stock entries are zero), the order is required to be returned to the customer marked "unmet", with some suitable covering letter; the order instance is deleted, but the customer information is retained. Again, the detail is not pursued here.

5 Top-level generic specification

This section presents a top-level Z specification, such as may be used to derive and clarify requirements. It demonstrates the scope of the system and outlines its structure. The Z description focuses on the state and operations of the system. It excludes low-level information, such as attributes and detailed operation pre- and post-conditions. The approach is derived from the SAZ method, in that the starting point is a data model and the formal description expresses the entities and relationships from that model[1].

Even with a predetermined approach and style, there are a number of possible representations in Z for the system described. One approach might be to represent each entity such as *Order* as a set of instances, and each relationship as a relation between the instances of the entities involved. Here, a less abstract but more intuitive approach is used: we model each entitiy as a mapping from identifier to instance. This removes the need to modify the underlying structure using a formal refinement of the specification.

In most published Z case studies, *given sets* are used to achieve the required level of abstraction. A given set provides a pool of elements with no internal structure, and is a suitable model in a specification that abstracts away from such internal structure. However, in the development of a system, the high-level description needs to be capable of elaboration as further information about the data and processing structures emerges during development. The traditional approach is to refine a given set to a set of structured elements with the desired low-level properties. Such a refinement requires a proof to show that it has been performed correctly, which in turn constrains the kind of instantiations that can be made. Instead of using given sets to model entities, we present a description made up almost entirely of generic schemas.

5.1 Unique identifiers

In our high level Z representation, the data model entities that make up the state are modelled as functions from some unique identifier to the instance. Although

[1] There is no attempt to check that the formal and informal descriptions are equivalent, since it is the formal development that is of interest here. A true development method would need to document the extent of equivalence of the descriptions in the different notation and levels of abstration.

these identifiers could be introduced as generic parameters, they are in fact genuinely uninteresting. They have but one role: to provide a unique identity for otherwise potentially identical instances of an entity. The only interest in an identifier arises at implementation, when uniqueness must be guaranteed. We do not consider implementation here, so we model these identifiers with given sets[2].

$$[CID, OID, OLID]$$

CID are customer identifiers, *OID* are order identifiers, *OLID* are order line identifiers. (We choose not to specify the *BOOK* component yet: that is introduced to illustrate system extension in section 7.2.)

5.2 System state

Rather than specify the entire state space in one chunk, we split it into logically separate components [2, chapter 11]. Dotted lines (optional relationships) on the ERD are good places to think about splitting the state.

The customer entity is independent of the others. We model it using a simple schema, mapping customer identifiers to customer instances, generic in the type of those instances[3].

$$\begin{array}{|l}
\textit{CustomerBase}[CUST] \\
\hline
customer : CID \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$\mid\mid$\hidewidth}} CUST \\
\hline
\end{array}$$

As described in the ERD and text, an order must have some order lines; an order line must be related to one order. The order and order line entities are thus dependent, as defined by the required relationship *isPartOf* between the respective identifiers.

$$\begin{array}{|l}
\textit{OrderBase}[ORDER, OLINE] \\
\hline
order : OID \nrightarrow\!\!\!\!\!\!\! + ORDER \\
orderLine : OLID \nrightarrow\!\!\!\!\!\!\! + OLINE \\
isPartOf : OLID \leftrightarrow OID \\
\hline
isPartOf \in (\operatorname{dom} orderLine) \twoheadrightarrow (\operatorname{dom} order) \\
\hline
\end{array}$$

To capture the meaning of the ‘crows foot’ on the *isPartOf* relationship, we constrain the Z relation to be a total surjection (an order line instance is related

[2] In Z, given sets are disjoint. This may constrain their use to some extent, requiring the specification of a given set as a super set, and then more detailed subsetting. For any particular given set, such subsetting can be achieved using a free type. Z does not support generic free types, however.

[3] One difference in style between the use of generic definitions and generic schemas is the naming of the generic formal parameters. In the former, a parameter tends to be a single letter; in the latter it tends to be a short word indicating its role. The longer names lead to formatting problems in the case of multiple parameters.

to no more than one order instance, hence functional; each order line instance is related to an order instance, hence total; each order instance is related to at least one order line, hence surjective).

The full system state includes these two substate components, and the *isMadeBy* relationship between them.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{OrderingSystem}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}]} \\ \hline
\mathit{CustomerBase}[\mathit{CUST}] \\
\mathit{OrderBase}[\mathit{ORDER}, \mathit{OLINE}] \\
\mathit{isMadeBy} : \mathit{OID} \longleftrightarrow \mathit{CID} \\ \hline
\mathit{isMadeBy} \in (\operatorname{dom} \mathit{order}) \longrightarrow (\operatorname{dom} \mathit{customer}) \\ \hline
\end{array}
$$

The *isMadeBy* relation is constrained to be a total function (each order instance is related to precisely one customer instance, hence functional and total; but not necessarily all customer instances take part, hence not surjective).

5.3 Sample operations

The events that affect this small system are the receipt and deletion of orders, and the creation, modification and deletion of customers. (The processing of an order is not described, because the book description has been omitted.) Generic operations are defined to model these events. (In the examples below, we omit the error case schemas, for brevity.)

Having specified the state as independent substates, it is useful to define schemas for updating just these substates [2, chapter 10]. When we update on the customer substate, we do not change the *isMadeBy* relation; when we update the order part, we may change this relation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Delta \mathit{CustomerBase}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}]} \\ \hline
\Delta \mathit{OrderingSystem}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\Xi \mathit{OrderBase}[\mathit{ORDER}, \mathit{OLINE}] \\ \hline
\mathit{isMadeBy}' = \mathit{isMadeBy} \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Delta \mathit{OrderBase}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}]} \\ \hline
\Delta \mathit{OrderingSystem}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\Xi \mathit{CustomerBase}[\mathit{CUST}] \\ \hline
\end{array}
$$

Customer creation is straightforward. The customer details (for the moment simply a generic parameter) are input, and the new customer identifier is output.

$$
\begin{array}{|l}
\hline
\mathit{CreateCustomer}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\Delta \mathit{CustomerBase}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\mathit{cust}? : \mathit{CUST} \\
\mathit{cid}! : \mathit{CID} \\
\hline
\mathit{cid}! \notin \operatorname{dom} \mathit{customer} \\
\mathit{customer}' = \mathit{customer} \cup \{\mathit{cid}! \mapsto \mathit{cust}?\} \\
\hline
\end{array}
$$

When an order is created, the appropriate order and order line entity instances are created, and the relationship between their identifiers, and between the order identifier and the identifier of the customer placing the order, are updated. This could all be specified in a single operation, say by having the input include a sequence of the order lines that comprise the order. However, that leads to a relatively complicated definition that is not reusable for the purpose of adding a single order line. So we model the effect of the complete operation in two parts. The first creates an order with one attached order line[4]. The second adds an order line to an existing order.

$$
\begin{array}{|l}
\hline
\mathit{CreateOrder}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\Delta \mathit{OrderBase}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\mathit{orderLine}? : \mathit{OLINE} \\
\mathit{order}? : \mathit{ORDER} \\
\mathit{cid}? : \mathit{CID} \\
\mathit{olid}! : \mathit{OLID} \\
\mathit{oid}! : \mathit{OID} \\
\hline
\mathit{cid}? \in \operatorname{dom} \mathit{customer} \\
\mathit{oid}! \notin \operatorname{dom} \mathit{order} \\
\mathit{olid}! \notin \operatorname{dom} \mathit{orderLine} \\
\mathit{order}' = \mathit{order} \cup \{\mathit{oid}! \mapsto \mathit{order}?\} \\
\mathit{orderLine}' = \mathit{orderLine} \cup \{\mathit{olid}! \mapsto \mathit{orderLine}?\} \\
\mathit{isPartOf}' = \mathit{isPartOf} \cup \{\mathit{olid}! \mapsto \mathit{oid}!\} \\
\mathit{isMadeBy}' = \mathit{isMadeBy} \cup \{\mathit{oid}! \mapsto \mathit{cid}?\} \\
\hline
\end{array}
$$

[4] We cannot create an order with no order lines, because of the surjectivity requirement. Now would be a good time to check if that requirement is too strong.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AddOrderLine}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}]} \\
\Delta \mathit{OrderBase}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\mathit{orderLine}? : \mathit{OLINE} \\
\mathit{oid}? : \mathit{OID} \\
\mathit{olid}! : \mathit{OLID} \\
\hline
\mathit{oid}? \in \operatorname{dom} \mathit{order} \\
\mathit{olid}! \notin \operatorname{dom} \mathit{orderLine} \\
\mathit{order}' = \mathit{order} \\
\mathit{orderLine}' = \mathit{orderLine} \cup \{\mathit{olid}! \mapsto \mathit{orderLine}?\} \\
\mathit{isPartOf}' = \mathit{isPartOf} \cup \{\mathit{olid}! \mapsto \mathit{oid}?\} \\
\mathit{isMadeBy}' = \mathit{isMadeBy} \\
\hline
\end{array}$$

There are several operations to change the details of entity instances. These include an operation to select a specific customer (by identifier) and change the value of the instance that it identifies, and an operation to change the value of an instance of customer however many identifiers it has linked to it. The former is illustrated. The customer's relationships do not change.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ChangeCustomer}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}]} \\
\Delta \mathit{CustomerBase}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\mathit{cust}? : \mathit{CUST} \\
\mathit{cid}? : \mathit{CID} \\
\hline
\mathit{cid}? \in \operatorname{dom} \mathit{customer} \\
\mathit{cid}? \mapsto \mathit{cust}? \notin \mathit{customer} \\
\mathit{customer}' = \mathit{customer} \oplus \{\mathit{cid}? \mapsto \mathit{cust}?\} \\
\hline
\end{array}$$

Deletion is similar to modification, with the stronger precondition that a customer instance cannot be deleted if there are still orders for that customer.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DeleteCustomer}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}]} \\
\Delta \mathit{CustomerBase}[\mathit{CUST}, \mathit{ORDER}, \mathit{OLINE}] \\
\mathit{cid}? : \mathit{CID} \\
\hline
\mathit{cid}? \in \operatorname{dom} \mathit{customer} \setminus \operatorname{ran} \mathit{isMadeBy} \\
\mathit{customer}' = \{\mathit{cid}?\} \mathbin{\lhd\!\!\!-} \mathit{customer} \\
\hline
\end{array}$$

Similar operations can be defined for the other entities.

5.4 Discussion

The above Z specification is a simple, but precise, account of (part of) an abstract ordering system. It captures the essential relationships between entities without discussing any internal structure of those entities. (It captures the ERD, but no details from the underlying Data Dictionary.) It specifies how those relationships may be modified, incorporating all constraints that can be expressed without reference to internal entity structure.

Such a specification can be used as a starting point for discussion with customers, for example in a discussion of business rules or operational details for the system. Additional information is added in the next development phase, which is at a lower level of abstraction.

This first Z document has many possible instantiations, representing different development scenarios and customer requirements. Although it could be instantiated at this stage, there is nothing to be gained by doing so; the abstraction level requires no extra details, and the instantiation would simply use given sets for the actual parameters.

6 Design by Instantiation

The full development process is likely to be iterative; this case study merely captures one step in the development.

The objective of this design is to record the logical details of data and processing. When considering a formal development, the static and dynamic constraints are an area of particular interest. This is where formal approaches add most value for limited effort, compared to the traditional diagram-and-text models, which are generally poor at recording and exploring these system rules [19, 20].

The system description is constrained only in so far as the development outcomes are genuinely determined at this phase of the development. The specification of data must capture known client requirements. It is a matter of policy as to how such a specification captures additional data formatting and constraints; managers of development projects should determine whether requirements in this area are to be expressed at this stage.

As noted earlier, traditional Z development uses given sets for data that has not yet been fully defined. When the data types are elaborated, a refinement is required to move from given sets to the new types. Here we show how instead our generic formulation can be instantiated, to provide a first specialisation of these data types. In section 6.1 we illustrate the instantiation process by giving a 'traditional'-style given sets instantiation of the *CustomerBase*. In section 6.3 we show how the generic approach may be used at each level.

6.1 Traditional specification of the customer entity

The Data Dictionary says that a customer has a name, an address, the date when they become a customer, and a credit limit. The credit limit must be within some globally imposed limit. An address, in turn, comprises a house identifier, a street, a town, and a postcode. Furthermore, a house identifier may be a number or a house name.

First we specify the house identifier, using a free type. Simple free types, enumerated types, allow the specifier to make clear the specific values that an attribute can take, for example for status-check attributes, and for use in the processing specification. This is a step that is often overlooked in structured method developments, where the specifier may concentrate too early on the potential implementations of enumerated attribute domains. Enumerated types are often modified in the course of the development, as more or fewer statuses become necessary[5].

$$[\mathit{HOUSENAME}]$$

$$\mathit{HOUSE} ::= \mathit{number}\langle\langle \mathbb{N} \rangle\rangle \mid \mathit{name}\langle\langle \mathit{HOUSENAME} \rangle\rangle$$

Where a domain for a particular attribute is a subset of a wider set, it is given a specific domain name as an abbreviation for the wider. This acts as a reminder that there may need to be more detailed specification of constraints on the domain. So we introduce *limit*, which may need to be further constrained, for example, to be within some global limit.

$$\mathit{limit} == \mathbb{N}_1$$

$$\mid\ \mathit{maxLim} : \mathit{limit}$$

Schema types are used where some structural information is already known about a type. A typical example concerns the format of an address.

$$[\mathit{STREET}, \mathit{TOWN}, \mathit{POSTCODE}]$$

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{Address0}} \\
\mathit{house} : \mathit{HOUSE} \\
\mathit{street} : \mathit{STREET} \\
\mathit{town} : \mathit{TOWN} \\
\mathit{postcode} : \mathit{POSTCODE} \\
\hline
\end{array}$$

We are now in a position to specify the customer entity.

$$[\mathit{DATE}, \mathit{NAME}]$$

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{Customer0}} \\
\mathit{name} : \mathit{NAME} \\
\mathit{address} : \mathit{Address0} \\
\mathit{creditLimit} : \mathit{limit} \\
\mathit{registeredDate} : \mathit{DATE} \\
\hline
\mathit{creditLimit} \leq \mathit{maxLim} \\
\hline
\end{array}$$

[5] In Z, enumerated types must have unique values; thus, a value such as *notFound* cannot be part of more than one enumerated type. This can cause frustration in large specifications; however, the bonus is that, in complex operations, the status or message information is clearly readable.

The traditional Z style would then instantiate the customer substate as follows, but would eventually require the elaboration of most of its component types.

$$TradCustomerBase == CustomerBase[Customer0]$$

One advantage of this specification is that the Z toolkit operators for integers can be used when putting constraints on credit limits. The main disadvantage is that types become established, and developers are not prompted to consider how best features such as limits and maxima should be implemented.

6.2 Operator definition

In what follows, we use generic types to represent quantities such as credit limit. We still want to be able to specify that the limit must be less than some global maximum. So we generically specify the set of all total orders (that is, reflexive, antisymmetric, transitive and total), and we specify our generic comparison operator to be one such total order[6] [7].

relation $(\_ \preccurlyeq \_)$

$$\begin{array}{l}
[X] \\
\hline
\_ \preccurlyeq \_ : \{ r : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq r \land r \cap r^{\sim} \subseteq \mathrm{id}\, X \\
\qquad\qquad \land r \,⨾\, r \subseteq r \land r \cup r^{\sim} = X \times X \} \\
\hline
\end{array}$$

The use of a defined order avoids the use of meaning-free type operators that rely on the semantic understanding of the operator name[8], or the completely unacceptable provision of general purpose type operators (for example, a general *dateComparison* operator).

6.3 Generic specification of the customer entity

Continuing our style of using generics to model as yet unelaborated data types, we can use generic parameters in place of many of the above data types, depending on the extent to which type details are determined by the developers and clients.

[6] Note that this is a loose generic definition. Spivey Z has a proof obligation that a generic definition is uniquely determined for all possible instantiations. Standard Z permits loose generics; the proof obligation is that the definition is well formed at each point of instantiation.

[7] This may all seem a little over-complicated. But consider the case of dates or times. We do not want to be forced to model a date as a simple number, neither do we want to be forced to model it as a complicated structure yet. But we certainly want to be able to say one date is before another. Using a generic order, and requiring date to be ordered, solves the problem.

[8] The problem has not been solved entirely, however. An unwanted instantiation of X and $\_ \preccurlyeq \_$ is $\mathbb{N}$ and $\_ \geq \_$. A validation process is always required at instantiation.

Assuming that the address structure is accepted, this can be re-expressed as

$$\begin{array}{|l}
\multicolumn{1}{l}{Address[HOUSE, STREET, TOWN, POSTCODE]} \\
house : HOUSE \\
street : STREET \\
town : TOWN \\
postcode : POSTCODE \\
\hline
\end{array}$$

A generic limit is defined.

$$\begin{array}{|l}
\multicolumn{1}{l}{[LIMIT]} \\
maxLimit : LIMIT \\
\hline
\end{array}$$

The customer type is defined as a generic, incorporating elaborated types[9].

$$\begin{array}{|l}
\multicolumn{1}{l}{Customer[NAME, HOUSE, STREET, TOWN, POSTCODE, LIMIT, DATE]} \\
name : NAME \\
address : Address[HOUSE, STREET, TOWN, POSTCODE] \\
creditLimit : LIMIT \\
registeredDate : DATE \\
\hline
creditLimit \preccurlyeq maxLimit \\
\hline
\end{array}$$

The order and order line entities can be similarly defined.

$$\begin{array}{|l}
\multicolumn{1}{l}{Order[DATE]} \\
orderDate : DATE \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{[AMOUNT]} \\
minAmount : AMOUNT \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{OrderLine[AMOUNT, NOTE]} \\
quantity, supplied : AMOUNT \\
note : NOTE \\
\hline
minAmount \preccurlyeq quantity \\
supplied \preccurlyeq quantity \\
\hline
\end{array}$$

[9] It would be nice if Z had some support for grouping the generic parameters, to highlight the fact that some are relevant only to the further instantiation of *Address*, for example.

6.4 Instantiating the state

The instantiation uses the entity types to define all necessary sets and relationships. The full state includes the full list of generic parameters.

$$\begin{array}{l} \mathit{SystemI}[\mathit{NAME}, \mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \mathit{POSTCODE}, \mathit{LIMIT}, \\ \qquad \mathit{DATE}, \mathit{AMOUNT}, \mathit{NOTE}] == \\ \quad \mathit{OrderingSystem}[\mathit{Customer}[\mathit{NAME}, \mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \\ \qquad\qquad \mathit{POSTCODE}, \mathit{LIMIT}, \mathit{DATE}], \\ \qquad \mathit{Order}[\mathit{DATE}], \mathit{OrderLine}[\mathit{AMOUNT}, \mathit{NOTE}]] \end{array}$$

Substates may also be defined by instantiation, and constraints can be added if necessary. Although the quantity of generic parameters in this expression is unwieldy, this has an advantage in terms of traceability. Since the innards of the system types are explicit, it is clear what needs elaborating at a later stage, and where each component is used in the substates. This is analogous to an automatic, in-line indexing, which could form the basis for a development documentation tool.

6.5 Operations

Operations are also specified by instantiation, both of state components, and of the operations. However, most operations require elaboration of pre- and post-conditions, taking account of the greater state information, and research into business rules. A developer seeking additional predicates should, for example, be encouraged to check the attributes of all the entities in the specification, looking for range constraints, default entry values, derivation formulae, and relationships to the values of other attributes in the system.

The specification of an operation is illustrated for the creation of a customer.

$$\begin{array}{l} \mathit{CreateCustomerI}[\mathit{NAME}, \mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \mathit{POSTCODE}, \mathit{LIMIT}, \\ \qquad\qquad \mathit{DATE}, \mathit{AMOUNT}, \mathit{NOTE}] \\ \hline \mathit{CreateCustomer}[\mathit{Customer}[\mathit{NAME}, \mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \\ \qquad\qquad \mathit{POSTCODE}, \mathit{LIMIT}, \mathit{DATE}], \\ \quad \mathit{Order}[\mathit{DATE}], \mathit{OrderLine}[\mathit{AMOUNT}, \mathit{NOTE}]] \\ \mathit{today}? : \mathit{DATE} \\ \hline \mathit{cust}?.\mathit{registeredDate} = \mathit{today}? \\ \text{“Additional predicates to enforce state invariants”} \\ \mathit{cust}?.\mathit{creditLimit} \preccurlyeq \mathit{maxLimit} \\ \hline \end{array}$$

Notice that this operation has additional preconditions. So some attempted uses of the operation for certain values of *Customer* will fail, where the abstract operation succeeds for any value of generic *CUST*. Along with elaborating the data type, we have *strengthened the precondition*, and so we do not have a formal refinement (which permits only weakening the precondition). Generic instantiation has allowed us to separate concerns. The preconditions that depend on details of the entity structure are omitted until that entity is elaborated: a different elaboration could result in a different precondition. This permits a more abstract top-level specification, where the concrete instantiations are not classic refinements.

6.6 Discussion

The abstraction level of the specification presented here is similar to the level of abstraction achieved in published case studies that use integrations of Z with a structured technique or method. In the integrated methods area, it has generally been the case that the formal system description has been derived once, from data models (with dynamic and/or process models as relevant) with low-level documentation[10].

There are intuitive or aesthetic arguments in favour of specification by instantiation. There is a clear separation of concerns; the developer is encouraged to think abstractly and not to make premature decisions about implementation or design features.

However, the utility of the approach comes down to an argument between readability, or adaptability of formal descriptions, and simplicity in the formal structures used. The use of generics increases the complexity of the Z descriptions. Indeed, the assistance provided by support tools has, in the authors' experience, been jeopardised where proofs are required of features of descriptions that contain such nested generic instantiations.

7 Reusing and Elaborating through instantiation

There would seem to be a number of advantages to arriving at this level of description via instantiation.

- It is easy to modify the initial analysis-derived description, for example to amend the scope of the system developed, since the essential data and processing structures are not lost among lower-level details.
- It is easy to modify the data domains where a client corrects the analysts' interpretation or changes their mind about domains, or where a developer is required to produce another system with a similar structure but different details. Processing descriptions can be modified to include pre- and post-conditions, not as a refinement, but as a description of a different specification that can be derived from the analysis.

These advantages are further illustrated by changing the detail of the instantiation, both in terms of the data dictionary, and in terms of the static structure of the system.

[10] See for example, SAZ case studies [8, 19, 10], Semmens et al [21], Josephs et al [22].

7.1 Data dictionary modifications

The most common changes to a specification during the development of a system concern the details of data and constraints. In structured modelling, these are generally held in some form of textual data dictionary, and support the diagrammatic models. Formal specification is particularly clear in the areas of data domains and static and dynamic constraints; the generic form can easily adapt to capture alterations made. The problem reduces to a versioning issue, which is beyond the scope of this paper.

To illustrate the accommodation of such changes, consider the modifications,

- there are no more than n customers in the system;
- orders met in full have a marker attribute set.

These changes do not affect the high-level generic state, which captures the structure of the system (section 5); they affect only the details of the first instantiation of the structure, section 6.

First, the order entity type is modified to add the new attribute.

$$
\begin{array}{|l}
\hline
\mathit{RevOrder}[\mathit{DATE}, \mathit{MARKER}] \\
\mathit{orderDate} : \mathit{DATE} \\
\mathit{metInFull} : \mathit{MARKER} \\
\hline
\end{array}
$$

This specification replaces *Order* in subsequent instantiations. None of the operations on order need modifying, since none makes explicit reference to the component data attributes of the order. In general, however, an operation that made explicit reference to the component data attributes of a changed entity would require modification.

The constraint on the number of customers can be introduced as an elaboration to the state schema.

$$
\begin{array}{|l}
\mathit{customerLimit} : \mathbb{N}_1
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{RevOrderSystem}[\mathit{NAME}, \mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \mathit{POSTCODE}, \mathit{LIMIT}, \\
\qquad\qquad\qquad\qquad \mathit{DATE}, \mathit{AMOUNT}, \mathit{NOTE}] \\
\mathit{OrderingSystem}[\mathit{Customer}[\mathit{NAME}, \mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \\
\qquad\qquad \mathit{POSTCODE}, \mathit{LIMIT}, \mathit{DATE}], \\
\quad \mathit{Order}[\mathit{DATE}], \mathit{OrderLine}[\mathit{AMOUNT}, \mathit{NOTE}]] \\
\hline
\#\mathit{customer} \leq \mathit{customerLimit} \\
\hline
\end{array}
$$

Since the signature of the schema is unchanged, there are no knock-on effects in the operations. However, there is a new pre-condition in *AddCustomer*: a new customer cannot be added if it would take the system over the newly-imposed limit. It is a matter of style whether this pre-condition is left implicit or made explicit.

7.2 Structural modifications

During development of a system, it may be necessary to perform some extension or specialisation. The most obvious illustrations of this are the addition of an entity, and the subtyping of an entity in the system data model or state. Although fundamental changes to the data structure would require respecification, specialisation and extension to the model can generally be accommodated more simply. This promotes the reuse of formal descriptions.

Extending the state: The scenario data model (figure 1) shows an additional entity, $BOOK$. This can be defined generically and added to the system state. Side issues are the addition of the relationship with existing entities and the expression of any new constraints on the structure.

$[BID]$

$$\begin{array}{|l}
\multicolumn{1}{l}{BookState[BOOK]} \\ \hline
book : BID \text{ ⇻ } BOOK \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{BookOrderSystem[CUST, ORDER, OLINE, BOOK]} \\ \hline
OrderingSystem[CUST, ORDER, OLINE] \\
BookState[BOOK] \\
refersTo : OLID \leftrightarrow BID \\ \hline
refersTo \in (\mathrm{dom}\, orderLine) \longrightarrow (\mathrm{dom}\, book) \\
\forall\, o : \mathrm{ran}\, isPartOf \bullet (isPartOf^{\sim})\llparenthesis\{o\}\rrparenthesis \lhd refersTo \in OLID \text{ ⤔ } BID \\ \hline
\end{array}$$

The additional predicate state that no two order lines of an order may refer to the same book. Operation schemas need amending accordingly.

Specialising entities: Specialisation is illustrated by subtyping the customer entity to express two different kinds of customer:

$$\begin{array}{|l}
\multicolumn{1}{l}{Corporate[HOUSE, STREET, TOWN, POSTCODE]} \\ \hline
invoiceAddress : Address[HOUSE, STREET, TOWN, POSTCODE] \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{[RATING]} \\ \hline\hline
minRating : RATING \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Private[RATING]} \\ \hline
creditRating : RATING \\ \hline
minRating \preccurlyeq creditRating \\ \hline
\end{array}$$

The customer type is now composed of the common elements (defined in the original specification), and optional components of these types[11]. A predicate can require that a customer is of one subtype only. (The optionality mechanism is defined in section 2.1.)

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\mathit{SpecialCustomerSpec}[\mathit{NAME}, \mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \mathit{POSTCODE},} \\
\multicolumn{1}{l}{\qquad\qquad\qquad\qquad \mathit{CREDITLIMIT}, \mathit{DATE}, \mathit{RATING}]} \\
\mathit{Customer}[\mathit{NAME}, \mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \mathit{POSTCODE}, \\
\qquad\qquad \mathit{CREDITLIMIT}, \mathit{DATE}] \\
\mathit{private} : \mathit{optional}[\mathit{Private}[\mathit{RATING}]] \\
\mathit{corporate} : \mathit{optional}[\mathit{Corporate}[\mathit{HOUSE}, \mathit{STREET}, \mathit{TOWN}, \mathit{POSTCODE}]] \\
\hline
\#\mathit{private} = 1 \Leftrightarrow \#\mathit{corporate} = 0 \\
\hline
\end{array}$$

Selectors for private and corporate customers can be written [5]. Again, operations need extending and modifying accordingly.

8 Discussion

Traditional Z development uses given sets for initially unstructured data types that can be elaborated during the development. It relies on formal development methods such as refinement to move towards an implementation goal.

In contrast, we describe an approach that uses generic specification with elaboration by instantiation. It has the following properties:

- Separation of data and relationships. The abstract specification captures relationships between entities as captured by the ERD; the generic parameters are instantiated with structures defined from the data dictionary. Each can be modified independently of the other.
- Elaboration by instantiation. No proof obligations are generated by the instantiation. Different instantiations of the same specification can be used to produce different systems. Preconditions on operations can be introduced at the appropriate level of abstraction; development remains valid even though such precondition strengthening does not follow the formal refinement relationship.

It is notoriously difficult to document a development. Features that are developed successively to an implementation are scattered across models and within models, and traceability becomes a major problem. Whilst the generic development presented does not contribute any large-scale improvement in this area, the inclusion of all the required types in the headings of the generic schemas at least ensures that these are all available to the developer without searching the document for given sets and other type instantiations.

[11] This slightly clumsy formulation, or something equivalent, is needed in the absence of generic free types.

The case study example is small, and although we assert that the approach scales, it is not entirely clear how the levels of complexity introduced affect the readability and usability of the development products. The approach would definitely be improved by better Z support for

- formatting long lists of generic formal parameters
- grouping generic parameters
- generic free types

A tool to support a rigorous development following the approach described here (as opposed to a tool to support the formal descriptions and proofs) might be expected to have at least the following characteristics:

- discrimination between levels of detail in the development descriptions, encouraging the developer to work consistently through lower levels of abstraction (but, to support practical working, not necessarily requiring the developer to complete one level of abstraction before attempting a lower level);
- good visualisation and expansion of generic structures and generic instantiations, allowing the developer to explore structures, construct proofs and so on, without resorting to a complete expansion of all schemas;
- support for the development and use of templates, where a design follows the outline of the specification but with different structural details, or where routine structures such as entity sets, refinement retrievals, and structural proof conjectures, are required.

Support tools might also take design on to generation of, for example, relations for a relational database, or, at a less specific level, might provide guidance on the form of design and specification specialisation needed for different media (programming or database paradigms).

Acknowledgements

The work described in this paper derives in part from work undertaken during and as a result of the SAZ project, 1990–94, EPSRC grants (GR/J81655) and (GR/F98642).

We would like to thank Sam Valentine and Ian Toyn for helpful discussions.

References

1. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd Edition, 1992.
2. R. Barden, S. Stepney, D. Cooper. *Z In Practice.* Prentice Hall, 1994.
3. B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 2nd Edition, 1996.
4. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof* Prentice Hall, 1996.

5. M. d'Inverno and M. Priestley. Structuring Specification in Z to Build a Unifying Framework for Hypertext Systems. *ZUM'95: The Z Formal Specification Notation; Proceedings of Ninth International Conference of Z Users, Limerick, Ireland, September 1995*, pp83–102. LNCS 967. Springer Verlag, 1995.
6. L. T. Semmens, R. B. France, and T. W. G. Docker. Integrated Structured Analysis and Formal Specification Techniques. *The Computer Journal*, vol 35, no 6, 1992.
7. R. B. France and M. M. Larrondo-Petrie. A Two-Dimensional View of Integrated Formal and Informal Specification Techniques. *ZUM'95: The Z Formal Specification Notation; Proceedings of Ninth International Conference of Z Users, Limerick, Ireland, September 1995*, pp434-448. LNCS 967. Springer Verlag, 1995.
8. F. A. C. Polack, M. Whiston, and P. Hitchcock. Structured Analysis – A Draft Method for Writing Z Specifications. *Proceedings of Sixth Annual Z User Meeting, York, Dec 1991.* Springer Verlag, 1992.
9. F. Polack, M. Whiston, and K.C. Mander. The SAZ Project: Integrating SSADM and Z. *Proceedings, FME'93 : Industrial Strength Formal Methods, Odense, Denmark, April 1993.* LNCS 670. Springer Verlag, 1993
10. F. Polack, M. Whiston, and K. C. Mander. *The SAZ Method Version 1.1.* York, YCS 207, Jan 1994.
11. I. Hayes. *Specification Case Studies.* Prentice Hall, 2nd Edition, 1992.
12. S. Stepney. A Tale of Two Proofs. *Proceedings, 3rd Northern Formal Methods Workshop, Ilkley, Sept 1998.* BCS-FACS, 1998.
13. S. Stepney, D. Cooper, and J. C. P. Woodcock. More Powerful Z Data Refinement: Pushing the State of the Art in Industrial Refinement. *ZUM'98 : 11th international conference of Z Users, Berlin.* LNCS 1493. Springer Verlag, 1998.
14. E. Boiten and J. Derrick. IO-Refinement in Z. *Proceedings, 3rd Northern Formal Methods Workshop, Ilkley, Sept 1998.* BCS-FACS, 1998.
15. R. Banach and M. Poppleton. Retrenchment: An Engineering Variation on Refinement. *Proceedings, B98, Montpellier, France.* LNCS 1393. Springer Verlag, 1998.
16. I. Hayes. Specification Models. *Z Twenty Years On: What Is Its Future?, Nantes, France, October 1995.* 1995.
17. I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter *Northern Formal Methods Workshop, Ilkley, UK, September 1995.* 1995.
18. CCTA. *SSADM Version 4 Reference Manual.* NCC Blackwell Ltd, 1990.
19. H. E. D. Parker, F. Polack, and K. C. Mander The Industrial Trial of SAZ: Reflections on the Use of an Integrated Specification Method. *Z Twenty Years On: What Is Its Future?, Nantes, France, October 1995.*
20. F. A. C. Polack and K. C. Mander. Software Quality Assurance Using the SAZ Method. *Proceedings of Eighth Annual Z User Meeting, Cambridge, June 1994.* Springer Verlag, 1994.
21. L. Semmens and P. Allen. Using Yourdon and Z: an Approach to Formal Specification. *Proceedings of Fifth Annual Z User Meeting, Oxford, Dec 1990.* Springer-Verlag, 1991.
22. D. Redmond-Pyle and M. Josephs. Enriching a Structured Method with Z. *Workshop on Methods Integration, 26 September 1991, Leeds*, (unpublished)
23. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.