Contents

Chapter 1

Z

Jonathan P. BOWEN

Oui, l'ouvre sort plus belle
D'une forme au travail
Rebelle,
Vers, marbre, onyx, émail.

[Yes, the work comes out more beautiful from a material that resists the process, verse, marble, onyx, or enamel.]

– Théophile Gautier (1811–1872) *L'Art*

1.1 Overview of the Z Notation

Z (pronounced 'zed') is a formal specification notation based on set theory and first order predicate logic. The mathematical notation is supported by a library of operators known as the 'Z toolkit', which is largely formally defined within the Z notation itself [ISO 02, SPI 01]. The operators have a large number of algebraic laws which aid in the reasoning about Z specification. As well as the mathematical notation, there is a '*schema*' notation to aid in the structuring of the mathematics for large specification by packaging the mathematical notation into boxes that may be used and combined subsequently.

There are many Z textbooks, some available online (see for example, [BOW 03, JAC 97, LIG 00, WOO 96]). A widely used reference book is also accessible online [SPI 01]. Z has subsequently undergone a lengthy international ISO standardization process culminating in 2002 [ISO 02], which could help in the development of further tools to support the notation. In particular, an open source *Community Z Tools* (CZT) initiative is underway, based around XML [MAL 05]. Z has been extended in a number of ways, especially with object-orienting features (e.g., Object-Z [SMI 00]). The theoretical basis of Z has been explored extensively (e.g., see [HEN 03]). A range of

case studies and a Z glossary may be found in [BOW 03].

1.1.1 The Process of Producing a Z Specification

Z is typically used in a modelling style [BOW 04] in which an *abstract state* is included, containing enough information to describe changes in state that may be performed by a number of *operations* on the system. Each of the operations defines a *relation* between a *before* and *after* version of the state. The state may contain *invariants* which are predicates relating the various components in the abstract state which should always apply regardless of the current state of the system.

An *initial state* is defined as a special case of the more general abstract state, with the addition of extra constraining predicates. The description of the system is then modelled by this initial state, followed by an arbitrary interleaving of the operations in any order, only limited by any *preconditions* imposed by individual operations.

Often operations are designed to be *total* (i.e., with a precondition of *true*) so that they can be applied in any situation. This is especially useful in maintenance of the implemented operations (which could typically be procedure calls, for example) since preconditions are not explicitly obvious in a program implementation and a maintainer unaware of such restrictions may be tempted to use the operation in an inappropriate situation.

In the case of Z, a good place to start the specification is by positing a possible abstract state to model the system. Inevitably this will have to be changed in the course of producing the rest of the specification (except in trivial cases) but that is part of the learning process by which knowledge and understanding of the system is gained.

Next some operations which may be performed on the system should be considered. Initially only the result of successful operations which perform the desired result with no problems should be formulated. The abstract state should be modified as required if some important aspect cannot be adequately modelled without it, always checking for the possible effect on other operations.

As the specification evolves, given sets and useful axiomatic or generic definitions can be assumed, then formally defined and added at the beginning of the specification. Errors reports in the case of unsuccessful operations should be considered and added. Some of these will normally be common across several operations in a specification of any size.

In practical specifications, it will be found that parts of the specification are repeated across groups of operations. It is often worthwhile factoring out these parts, presenting and explaining their purpose once, and then using them subsequently. This will considerably reduce the size of most large specifications and make their assimilation easier for the reader.

Total operations are normally formulated typically as a disjunction of the successful and, if required, a number of error cases. An appropriate error indication, normally as some form of output, is normally included depending on the requirements.

Finally (perhaps surprisingly) the initial state should be considered as a special case of the abstract state. Often the contents of much of the state is most easily considered to be empty or to have some fixed value at the start of the life of the system, but may be more loosely specified if the exact value is unimportant.

During the production of the specification, questions will inevitably be raised. These should be discussed within the design team, with other colleagues, or with the customer as appropriate, normally in that order, to resolve the issues. In the next section of this chapter, a Z specification is presented with some of these questions interspersed with the formal Z specification. Informal description of the formal specification is also included. This should be designed to reinforce the concepts presented in the Z specification, especially in relating it to the real world.

In a finished and polished Z specification, the informal annotation should normally be about the same length as the formal description. As a rule of thumb, it is a good idea to attempt to describe each line of predicate in Z schema boxes with a matching sentence of text written in a natural style. Ideally the informal part of the specification should be meaningful on its own, even if the formal part is removed. In fact this could be useful if the description is to be presented to a customer who may be unable to assimilate the Z specification itself.

1.2 Analysis and Specification of Case 1

Most specifications, formal or otherwise, are presented as a *fait accompli* after the specification has been produced, normally with no hint as to how the specification has been produced. There is some guidance on the use of formal methods in general but in the case of specific notations, even most textbooks tend to concentrate on finished specifications rather than the progress of specifications from initial concept (requirements) to completion.

In practice, the *process* of producing the specification can be as important as or even more important than the specification itself. The knowledge gained by the specifier in preparation before consideration of implementation details can be invaluable in resolving errors before the detailed design and subsequent stages, making them much cheaper to correct. In this section we consider typical questions posed during the specification process when using the Z notation for the first case study.

Question 1: What *given sets* are needed for the specification?

Answer: Z, as a typed language, provides the facility of including a number of distinct sets (called '*given sets*' or '*basic types*') for subsequent use in a specification. The sets are potentially infinite unless limited to being finite later in the specification. The set of integers $\mathbb{Z}$ is available in all Z specifications as part of the standard mathematical 'toolkit' library. Other given sets are normally discovered as a Z specification is formulated. Here we define sets of order identifiers and products which can potentially be held in stock:

$$[OrderId, Product]$$

The exact nature of the elements of these set is unimportant to the specification and is thus not elaborated further. An implementor would chose a specific representation for them in due course.

Question 2: What states can orders have?

Answer: The requirements mention two states, 'pending' and 'invoiced'.

We define a set *OrderState* with just two elements in it to model the states of *pending* and *invoiced* which an order can take as it progresses:

$$OrderState ::= pending \mid invoiced$$

Here *OrderState* is defined as a given set, but is limited to having two distinct elements, *pending* and *invoiced*, representing different possible states. Further states could be added later if that proves to be necessary.

Question 3: What *abstract state* is needed to model the system?

Answer: In Z, operations normally act on an abstract state, relating a *before state* to an *after state*. We need to model the state products in stock and orders including their invoicing status.

The quantity of each of the products in stock needs to be recorded, so a bag (also known as multiset) can be used to model this in a *vertical schema* called *Stock*:

Stock

$$stock : \mathrm{bag}\, Product$$

This includes a single *state component*, a bag called *stock* drawn from the set of bag *Product*. In Z, as with many programming languages, the ':' in declarations can be read as 'is a member of' like '$\in$' in predicates. Note that Z is case sensitive and many Z specifications use this in standard ways to help the reader. For example, here lower case names are used for state components and names starting with an upper case letter are used for given sets and schema names.

In the Z toolkit, the set of bags is defined as: $\mathrm{bag}\, X == X \pfun \mathbb{N}_1$, the set of partial functions between some set X and the strictly positive integers (greater than zero). This allows a record of the number of products in stock in the *Stock* schema above. For example, if *nuts* and *bolts* are valid products, then $stock = \{nuts \mapsto 5, bolts \mapsto 6\}$ would indicate that there are 5 *nuts* and 6 *bolts* in stock. $a \mapsto b$ is a graphic *maplet* notation used in Z to indicate the *Cartesian product* pair (a, b).

Question 4: Is it really required that an order be limited to a single type of product and an associated quantity or would a set of these be preferable?

Answer: The informal requirements indicate this, but it might be considered over-restrictive. A user may wish to order several types of product at once and this should be discussed with the customer. Here we assume that the customer decides to allow orders of one *or more* products for extra flexibility, but not empty orders (i.e., an order for no products).

Since *stock* is defined as a bag of products, it is convenient to define an order as a bag of products too. However, whereas the stock may be completely empty, an order must consist of one (or more) products:

$$Order == \{order : \mathrm{bag}\, Product \mid order \neq \varnothing\}$$

In the above, *Order* is defined using an *abbreviation definition* ('$==$') and *set comprehension* ('$\{\ldots \mid \ldots\}$'). Subsequently, any use of *Order* is the equivalent of using the right hand side of this definition directly. This is useful for expressions that are reused a number of times during a specification. The properties of the expression can be introduced in one place informally; the expression can be given a name formally and then used later as required. The constraint predicated after the '$|$' in the set comprehension above (which can be read as 'such that') normally limits the declaration(s) in some way (here to being non-empty).

The predicate constraint $order \neq \varnothing$ could also have been equivalently written as $\#order > 0$ or $\#order \geq 1$ where '$\#$' indicates the cardinality (size) of a set. If it was decided that only a single product is to be allowed we could write $\#order = 1$. This would allow us to easily change the specification subsequently if the customer changes his/her mind. We could even allow empty orders ($Order == \mathrm{bag}\, Product$). The rest of the specification can be left the same, whichever of these choices are made.

Continuing with the definition of the abstract model, the status of orders can be modelled as a function from an identifying *OrderId* to their state (*pending* or *invoiced*). State components *orderStatus* and *orders* are packaged into an *OrderInvoices* schema with appropriate *type* information:

OrderInvoices

$$orders : OrderId \pfun Order$$
$$orderStatus : OrderId \pfun OrderState$$

$$\mathrm{dom}\, orders = \mathrm{dom}\, orderStatus$$

Here, *orderStatus* and *orders* are *partial functions* from the set *OrderId*. The functions are partial (i.e., their domains do not necessary cover the whole of the *OrderId* set in this case) since only valid orders are mapped in this way.

All orders have a status associated with them. This type of general information that must apply at all times (whatever the specific state of the system at any given time) is presented as a state *invariant* predicate in most Z specifications (e.g., $\mathrm{dom}\, orders = \mathrm{dom}\, orderStatus$ above, constraining the domains of both functions to always be the same).

Question 5: Should order identifiers be unique for the entire lifetime of the system?

Answer: We could decide that new identifiers must never have been used previously or that they just need to be unique at any given time. The state specification so far assumes the former, which is easiest. However, if the latter is required, we must augment the state with further information on fresh new references that can be issued at any particular time.

The schemas *Stock* and *OrderInvoices* can be combined in a new *State* schema using *schema inclusion*, together with a further state component *newids*. The inclusion of *Stock* and *OrderInvoices* means all the declarations and associated predicates are available.

State

$$\begin{array}{|l}
Stock \\
OrderInvoices \\
newids : \mathbb{P}\, OrderId \\
\hline
\mathrm{dom}\, orders \cap newids = \varnothing \\
\hline
\end{array}$$

Question 6: What initial state is required for the system?

Answer: The requirements do not make this clear; if not defined in Z, the system could start in any valid state that satisfies any state invariants. Typically many state components are most usefully initialized to empty sets or some predetermined value. For example:

InitState

$$\begin{array}{|l}
State' \\
\hline
stock' = \varnothing \\
orders' = \varnothing \\
newids' = OrderId \\
\hline
\end{array}$$

The decoration '′' added to the *State* schema included above percolates through to all the state components declared in the schema (*stock*′, etc.). Note that all the predicates are combined using conjunction by default. The predicate $orderStatus' = \varnothing$ is implied because of the state invariant $\mathrm{dom}\, orders' = \mathrm{dom}\, orderStatus'$ from the *OrderInvoices*′ schema and hence can be omitted. All possible identifiers are available for use initially.

Question 7: Are there any constraints that apply for all operations on the system?

Answer: If so, they may be specified using the 'Δ' convention of Z:

Δ*State*

$$\begin{array}{|l}
State \\
State' \\
\hline
newids' = newids \setminus \mathrm{dom}\, orders' \\
\hline
\end{array}$$

Here an undashed *before state* (*State*) and a matching dashed *after state* (*State*′) are included.

If any new identifiers are used for orders (and hence their status) these are no longer available for use by any subsequent operation. Thus they are removed from the

set of new identifiers. Any operation including $\Delta State$ need not explicitly consider the value of $newids'$ since it will automatically be handled by the predicate in the schema above.

A change of state is specified using the $\Delta State$ schema convention. This defines a 'before' state *State* (which includes the four state component *stock*, *orderStatus*, *orders* and *newids* in this case) and an 'after' state $Invoices'$ which includes matching dashed state component ($stock'$, etc.).

Question 8: What operations are required?

Answer: Only a single operation to invoice an order seems to be required since many aspects do not have to be taken into account.

Question 9: What inputs and/or outputs are needed by the operation?

Answer: An input $id?$ is required to specify which invoice is to be updated. Note that in Z, a trailing '?' indicates an input and a trailing '!' indicates an output by convention.

Question 10: What *preconditions* apply?

Answer: In Z, preconditions are predicates in operations that apply only to before states and inputs. Preconditions may be calculated by existentially quantifying the after states and outputs, and then simplifying the resulting predicate. See later for an example.

For an order to be successfully invoiced, there must be enough stock available to fulfil the order and the status must be pending. These are *preconditions* that must be satisfied to change the order state to *invoiced*.

Question 11: What is the effect of the operation?

Answer: The effect of the operation is a relation between the before state and inputs with the after state and outputs, proving a *postcondition* for the operation. Often, although not always, this can be specified *explicitly* (e.g., in the form $stock' = \ldots$, etc. for all after state components and outputs). Indeed, checking for predicates in this form is a useful check to ensure that no important postconditions have been omitted. The lack of a predicate in this form for a particular after state component or output is not necessarily an indication of an error in the specification, but it is all too easy to omit a predicate of the form $x' = x$ when no change of state is required.

All this information discussed above is included formally in an *InvoiceOrder* operation as follows:

InvoiceOrder

$\Delta State$
$id? : OrderId$

$orders(id?) \sqsubseteq stock$
$orderStatus(id?) = pending$
$stock' = stock \cup orders(id?)$
$orders' = orders$
$orderStatus' = orderStatus \oplus \{id? \mapsto invoiced\}$

$\sqsubseteq$ is the sub-bag relational operator from the Z toolkit. As used in the schema above, this ensures a precondition that their are enough quantities of the required product(s) in stock. For example, $\{nuts \mapsto 3\} \sqsubseteq \{nuts \mapsto 5, bolts \mapsto 6\}$ is *true*.

Another precondition is that the status of the order must be *pending*. If the preconditions are satisfied, the required product quantities are removed from the available stock using the bag difference operator ('$\cup$', cf. the set difference operator '$\setminus$' for sets). Here for example, $\{nuts \mapsto 5, bolts \mapsto 6\} \cup \{nuts \mapsto 3\}$ would result in $\{nuts \mapsto 2, bolts \mapsto 6\}$.

The precondition $id? \in \mathrm{dom}\, orders$ could be included if an explicit check for $id?$ being a valid existing order identifier is required. This is also equivalent to $id? \in \mathrm{dom}\, orderStatus$ due to the invariant $\mathrm{dom}\, orders = \mathrm{dom}\, orderStatus$. However this precondition is implied by both the preconditions included in the *InvoiceOrder* implicitly since $id?$ is applied to *orders* and *orderStatus* using *function application*; this is only valid if $id?$ (in this case) is in the domain of the function. Here we decide to omit an explicit check for simplicity of presentation. However, consideration of this precondition as a separate case could affect the errors conditions returned by the complete operation (see later) and this should be discussed with the customer.

The orders themselves are unaffected by the operation above, as specified by $orders' = orders$. The order status is updated to *invoiced* using the *overriding* operator ('$\oplus$') from the Z toolkit. This operator is commonly used in Z specifications to update a small part of state components that are binary relations (often functions) in Z operation schema. Here the state of the maplet $id? \mapsto pending$ is replaced by a new maplet $id? \mapsto invoiced$, leaving the status of all other orders unchanged.

Question 12: What about error conditions?

Answer: Normally successful operations, where the preconditions are satisfied and the operation does what is required, are considered first in Z. The precondition can be calculated and the error condition(s) must have a precondition which handle the negation of this to eventually produce a *total operation* with a precondition of *true* (i.e., it can be invoked safely at any time) by combining the successful and error cases using disjunction.

Question 13: Are *error reports* required?

Answer: Nothing is said in the requirements, but most customers would wish to know if an operation was successful or not once it has been undertaken. They will

probably wish to know the nature of the error as will if more than one error is possible in a particular operation. Thus we define a set of possible reports from operations:

$$Report ::= OK \mid order\_not\_pending \mid not\_enough\_stock \mid no\_more\_ids$$

If further error reports prove necessary (e.g., if the system is upgraded later), they could be added to *Report* above as required subsequently. Above we define all error reports used in this chapter.

For successful operations, a suitable report is normally required to inform the user. Since this is a standard feature of successful operations, this can be separated out in a separate schema production an output report *rep*!:

Success

$$rep! : Report$$

$$rep! = OK$$

Question 14: What if the order state is not pending?

Answer: For error cases where the precondition does not hold, it is normal to assume the state is not to change. We define an error schema with a precondition that is the negation of one of the preconditions in the *InvoiceOrder* schema:

InvoiceError

$$\Xi State$$
$$id? : OrderId$$
$$rep! : Report$$

$$orderStatus(id?) \neq pending$$
$$rep! = order\_not\_pending$$

$\Xi State$ ensures that all the dashed state components in the after state are the same as the matching undashed state components in the before state; in this case, $stock' = stock \wedge \ldots$ Thus, the entire state afterwards is the same as the state before in the case of the error above.

Question 15: What if not enough stock is available for the order?

Answer: Here we return an alternative error report so the user can detect which error has occurred:

StockError

$$\Xi State$$
$$id? : OrderId$$
$$rep! : Report$$

$$\neg\, orders(id?) \sqsubseteq stock$$
$$rep! = not\_enough\_stock$$

Question 16: Should either error take priority if they both occur?

Answer: If so, an extra predicate giving the negation of the other error's precondition will be needed in one or other error schema above. If not, perhaps because the customer has no preference, this can be left non-deterministic. The decision can then be made by the implementor, depending on which is easiest, most efficient, etc., in the final design. It is good practice to leave design decisions to after the specification stage if they are not important at this point to give the design team as much freedom as possible in the implementation.

An error schema covering the case of $id? \notin \mathrm{dom}\, orders$ explicitly (i.e., the specified $id?$ is not a valid order in the system) could also be added if required by the customer, but we have omitted this case here for brevity. Instead one or other of the two errors that are included may be returned (non-deterministically) in this case.

A total operation for ordering where the precondition is *true* can now be specified:

$$InvoiceOrderOp == (InvoiceOrder \land Success) \lor InvoiceError \lor StockError$$

The above is a *horizontal schema* definition for a new schema *InvoiceOrderOp* in terms of a number of existing schemas. These are combined using schema operators, namely schema conjunction ('$\land$') and disjunction ('$\lor$'), based on the matching logical connectives. Both operators merge the state components of the schemas involved. Any components with the same name must be type-compatible (and are normally declared in an identical manner to avoid confusion). The predicates in the schemas involved are combined using logical conjunction or disjunction respectively.

Schema conjunction is normally used when building up a larger specification from smaller specification parts. Schema disjunction is normally used when specifying choice between two or more alternatives, typically successful and error operations. Normally any preconditions are disjoint to avoid any unexpected consequences. In a total operation, the disjunction of all the preconditions of the schema being combined is *true*.

If we do not have to take new orders, cancellations and addition to the stock into account, no other operations are required. However the precondition of the *InvoiceOrder* operation schema is such that the invoice must already be *pending* and there must be enough stock available to fulfil the order. Other operations are needed to make these true. Here we could assume that an arbitrary operation $\Delta State$ can be invoked at any time before *InvoiceOrderOp* operations.

In Z, exactly which schemas represent the abstract state, initial state and allowed operations is normally left informal and is just indicated in the accompanying text. There is no syntactic feature to distinguish these in Z, although some tools (e.g., the ZANS animator) have hidden directives to indicate these if required. In this particular example, the allowed operations is an area that would certainly need further discussion with the customer to avoid any misunderstanding.

1.3 Analysis and Specification of Case 2

Question 17: What extra operations are needed?

Answer: Assuming that Case 2 is an extension of Case 1, three further operations are indicated from the requirements to handle new orders, cancellation of orders and entries of quantities in the stock. However, these are not elaborated further.

Question 18: What inputs/outputs, preconditions and postconditions need to be included for an operation to handle new orders?

Answer: An order must be provided as an input and a valid fresh identifier is output by the operation. A new order leaves the stock unchanged but updates the orders and their status appropriately.

NewOrder

$\Delta State$
$order? : Order$
$id! : OrderId$

$id! \in newids$
$stock' = stock$
$orders' = orders \cup \{id! \mapsto order?\}$
$orderStatus' = orderStatus \cup \{id! \mapsto pending\}$

Note that *id*! is not explicitly set and can be any convenient new identifier. Here we assume that the status of the new order is *pending*; this should be discussed with the customer to check that this is what is actually required.

Question 19: When cancelling an order, is information concerning the order to be retained by the system?

Answer: We could either remove all information associated with the order from the system completely, or retain this information for possible future use. Here we assume that the information is no longer required, which is the simplest option, but this should be discussed with the customer. Perhaps some sort of auditing will be required of the system, including cancelled orders.

Question 20: What inputs/outputs, preconditions and postconditions are required for an operation to handle cancellations of orders?

Answer: Cancelling an order completely removes an existing order (determined by a valid order identifier input *id*?) from the system:

CancelOrder

$\Delta State$
$id? : OrderId$

$orderStatus(id?) = pending$
$stock' = stock$
$orders' = \{id?\} \lhd orders$
$orderStatus' = \{id?\} \lhd orderStatus$

The Z toolkit domain anti-restriction operator '$\lhd$' used above removes part of a relation (often a function) where the domain overlaps with a specified set. In the above example, a single element is removed in each case. We have assumed that the status of the order to be cancelled is *pending* as opposed to *invoiced* since this avoids problems of re-adding stock; this should be discussed with the customer. As for the *InvoiceOrder* operation previously, $id? \in \mathrm{dom}\, orders$ is implied.

Note that cancelled order identifiers can in fact be inferred as $OrderId \setminus (newsids \cup \mathrm{dom}\, orders)$ given the operation above. This could be useful if further requirements are added in the future.

Question 21: Is the finiteness of stock quantities (or any other state component for that matter) important?

Answer: Here natural numbers have been used for stock quantities and these are potentially infinite and hence of unbounded size in any corresponding implementation. Practical implementations will require some limit on the maximum size of stock, often determined by the system's computer architecture. If this is to be modelled in the specification, additional preconditions and error schemas will be required. In the specification below we assume no such requirements, but finiteness of state components is something that should always be discussed with the customer in practice.

Question 22: What inputs/outputs, preconditions and postconditions are required for an operation to handle entries of quantities in the stock?

Answer: Entering new stock can be effected using bag union:

EnterStock

$\Delta State$
$newstock? : \mathrm{bag}\, Product$

$stock' = stock \uplus newstock?$
$orders' = orders$
$orderStatus' = orderStatus$

The bag union operator ('$\uplus$') takes two bags and forms a new bag consisting of the sums of matching elements in these two bags (or just the elements in cases where there

is no match). For example, $\{nuts \mapsto 5, bolts \mapsto 6\} \uplus \{nuts \mapsto 3, washers \mapsto 1\}$ would result in $\{nuts \mapsto 8, bolts \mapsto 6, washers \mapsto 1\}$.

Here we assume that there is no limit on the amount of stock that can be held; in practice there may be a limit; this should be discussed with the customer and added as a precondition if appropriate.

Question 23: Are error reports required and to what level of detail?

Answer: Most customers will want operations to report errors and take appropriate action in these cases (typically although not always leaving the system state unchanged). The error report could simply be some status value or further information could be useful. Details of error handling are often omitted or glossed over in requirements documents, but should be discussed in detail with the customer before implementation. Producing a Z specification and calculating preconditions of successful operation is a good way to determine what errors are relevant to each operation. In Case 2, the following additional error reports are needed.

In the *NewOrder* operation $id! \in newids$ implies that $newids \neq \varnothing$. This is an example of an *implicit precondition* (i.e., a precondition that is not explicitly stated). Such preconditions can be found by formally calculating the precondition. This involves existentially quantifying the after states and outputs:

$$\begin{array}{l} \exists\, State'; id! : OrderId \bullet \\ \quad newids' = newids \setminus \operatorname{dom} orders' \land \\ \quad id! \in newids \land \\ \quad stock' = stock \land \\ \quad orders' = orders \cup \{id! \mapsto order?\} \land \\ \quad orderStatus' = orderStatus \cup \{id! \mapsto pending\} \end{array}$$

The *one-point rule* allows existentially quantified variables that occur once in the form '$x = \ldots$' to be eliminated, giving:

$$\exists\, id! : OrderId \bullet id! \in newids$$

Since for an element to be a member of a set, the set must be non-empty, this simplifies to:

$$newids \neq \varnothing$$

Because of this implicit precondition, the (perhaps unlikely) event of running out of new identifiers needs to be handled:

IdError

$$\begin{array}{|l} \Xi State \\ rep! : Report \\ \hline newids = \varnothing \\ rep! = no\_more\_ids \\ \hline \end{array}$$

Notice that the value of *id*! is not explicitly defined in the case of an error and thus could take on any value.

The total operations with appropriate reports can now be specified:

$$NewOrderOp == (NewOrder \land Success) \lor IdError$$
$$CancelOrderOp == (CancelOrder \land Success) \lor InvoiceError$$
$$EnterStockOp == (EnterStock \land Success)$$

Question 24: Are further operations such as status operations required?

Answer: It is often useful to have operations which return part of the state while leaving the system state unchanged. Once an abstract state for the modelling of the system has been formulated, this can be inspected and potentially useful status operations can be suggested to the customer. In this case, the state components comprise of *orderStatus*, *stock* and *orders*, and information on any of these could be returned.

1.4 Validation of the Specification

There are a number of checks that are worth performing on a Z specification once a draft has been formulated to reduce the number of errors it contains. For example:

- Check that the change of state for all components of the abstract state has been considered in every operation. It is easy to forget some parts of the state, in which case the meaning of the specification is that the after state for that component is totally unrelated to the before state and thus may take on any arbitrary value in an implementation. This is rarely what the customer wants in practice.
- Check that preconditions of successful and error parts of operations are disjoint in general. Otherwise there may be incompatibilities or potentially even a *false* specification otherwise.
- Check that preconditions of total operations are *true*. If they are not, there some cases that are not specified and which may be problematic in the implementation or subsequent maintenance.
- Check the specification type-checks using a mechanical type-checker. If the specification is not type-correct it is meaningless in a formal sense, although of course it can still impart some useful information to a human reader. A number of both free and commercial Z type-checkers have been produced (e.g., CADiZ, Formaliser, fUZZ and ZTC). It is recommended that all but the most trivial Z specifications should be mechanically type-checked. The Z text presented in this chapter has been type-checked using the fUZZ and ZTC tools.
- Attempt validation proofs to check the specification behaves as expected. If provable, these help in confirming the correct understanding and intuition of the specification; if they turn out to be false this may indicate a problem in the specification, or at least in the understanding of it. Mechanical tool support for

proofs in Z, such as Z/EVES, is available, but takes a significant amount of skill to use effectively.

- Animate the specification (e.g., using the ZANS animator associated with the ZTC type-checker). This can be useful to check the specification acts as expected, but will typically only work for 'explicit' and finite cases where the after state and output are defined explicitly and deterministically in terms of the before state and inputs. Normally a specification will need some adaptation to allow it to be animated. Nevertheless, this may prove to be a useful exercise in the removal of errors from the original specification. Indeed, ZANS reports whether operations are *explicit* (i.e., all the after state components and outputs are deterministically defined in terms of the before state components and inputs) and this is itself useful information for checking a specification.
  An alternative approach is to rapid-prototype the specification in a high-level programming paradigm, such as a logic or functional programming language. Prolog is a popular choice for rapid-prototyping Z specifications.

Note that a Z specification cannot be *verified* formally in general since there is (normally) no other mathematical description to verify it against. Typically requirements used to produce a formal specification are informal (e.g., natural language, diagrams, etc.), and this is certainly true in this case.

However it is possible to *validate* a Z specification by posing challenge hypotheses that are believed (and hoped) to be true for the intuition of the developer. Proving these to be true increases the confidence in the correctness of the specification (i.e., that the specification is what is required).

Checks on the consistency of the specification can also be formally undertaken as proofs. For example, the existence of an initial state for the entire system, or a post-state for each operation, can be checked. In general it is considered desirable in Z to specify total operations where the precondition is *true* (as demonstrated earlier in this chapter). The precondition for each operation can be formally calculated to check this (as done earlier for the *NewOrder* operation in Section 1.3).

Animation (attempting to execute the specification directly) or rapid-prototyping (producing an executable version of the specification with minimal development using a very high-level programming language, e.g., in the form of a logic program or a functional program) are additional approaches that help in the validation of the specification.

1.5 The Natural Language Description of the Specifications

The Z-style of specification dictates that the natural language description should accompany the formal Z text. This is what has been done in Sections 1.2 and 1.3 although extra didactic material has also been included.

Typically the informal description is of approximately the same length as the formal description, and certainly this is a good guideline to follow. It is a good aim to describe the system being specified in a form such that removal of the formal text would

still render an understandable informal document. Often it is found that producing a formal Z document results in a better, clearer, less ambiguous informal description of the system as well (e.g., for inclusion in a manual or for presentation to a customer).

1.6 Conclusion

Z is mainly used at the specification level. Some data and operation refinement towards an implementation is possible in Z [DER 01] but at some point a jump to code must be made, typically informally. A program is considered correct with respect to a Z specification operation if it can be run in more situations (the precondition is more relaxed) or if it is more deterministic (the postcondition is more strict). However, many Z operations already have a precondition of *true* (i.e., the operation is 'total' and can be used in any situation) and are often 'explicit' (i.e., the operation is deterministic). In the operations specified in this chapter, total operations have been provided. Most of the operations are explicit apart from the allocation of identifiers for new orders.

If an operation is invoked in a state where it is not defined, then anything can happen. It is typically not desirable, and is the reason why total operations are normally specified.

If significant formal development is required, it is normally better to use a notation designed for this, such as the Abstract Machine Notation (AMN) of the B-Method. However, many systems can cost-effectively benefit from formal specification alone, to help in avoiding the introduction of errors at the specification stage. In this case, Z is a very appropriate general purpose formal specification to use. Normally, formal *development* is much more expensive than formal *specification* and may only be worthwhile in very high-integrity systems [BOW 99].

The Z specification in this chapter was originally produced in less than a day. Problems, inconsistencies and misunderstandings have been resolved by the author alone on an ad hoc basis. A number of specific questions have been raised explicitly and possible different specifications presented. The next step in practice would be to discuss these with the customer to solve the issues; however this has not yet been done. Thus the case study specification as presented is a specification in the course of construction and perhaps has added interest for that reason.

For further online information on Z maintained by the author, see:

`http://vl.zuser.org/`

Bibliography

[BOW 03] Bowen J.P. Formal Specification and Documentation Using Z: A Case Study Approach, 2003. Originally published by International Thomson Computer Press, London, 1996. URL: `http://www.zuser.org/zbook/`

[BOW 99] Bowen J.P., Hinchey M.G. High-Integrity System Specification and Design. Formal Approaches to Computing and Information Technology series

(FACIT). Springer-Verlag, London, 1999. URL:
`http://vl.fmnet.info/hissd/`

[BOW 04] Bowen J.P., Hinchey M.G. Formal models. In: Tucker, Jr. A.B. (Ed.) Computer Science Handbook 2nd edition. Chapman & Hall / CRC, ACM, chapter 106, 106-1–106-25, 2004.

[DER 01] Derrick J., Boiten E.A. Refinement in Z and Object-Z. Formal Approaches to Computing and Information Technology series (FACIT). Springer-Verlag, London, 2001

[HEN 03] Henson M.C., Reeves S., Bowen J.P. Z logic and its consequences. CAI: Computing and Informatics, 22(4):381–415, 2003

[ISO 02] ISO/IEC Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics, ISO/IEC 13568:2002

[JAC 97] Jacky J. The Way of Z: Practical Programming with Formal Methods. Cambridge University Press, Cambridge, 1997. URL:
`http://staff.washington.edu/~jon/z-book/`

[LIG 00] Lightfoot D. Formal Specification Using Z 2nd edition. Palgrave, Basingstoke, 2000

[MAL 05] Malik P., Utting M. CZT: A framework for Z tools. In Treharne H., King S., Henson M., Schneider S. (eds.) ZB 2005: Formal Specification and Development in Z and B. Lecture Notes in Computer Science, volume 3455, 65–84. Springer-Verlag, Berlin, 2005

[SMI 00] Smith, G. The Object-Z Specification Language. Advances in Formal Methods series. Kluwer Academic Publishers, 2000. URL:
`http://www.itee.uq.edu.au/~smith/objectz.html`

[SPI 01] Spivey J.M. The Z Notation: A Reference Manual, 2nd edition, 2001. Originally published by Prentice Hall International Series in Computer Science, London, 1992. URL: `http://spivey.oriel.ox.ac.uk/~mike/zrm/`

[WOO 96] Woodcock J.C.P., Davies J. Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science, London, 1996. URL:
`http://www.usingz.com/`