

The use of Z to formally specify a commercial data processing system

P.R. Chudley

*Department of Computer Science, City
Polytechnic of Hong Kong, 83 Tat Chee Avenue,
Yau Yat Chuen, Kowloon, Hong Kong*

ABSTRACT

Although the number of organisations using formal specification techniques is expanding, its use tends to be focused upon safety critical systems, systems software and research. It is a common belief that formal specification techniques are not suitable, or too time consuming to use when developing commercial data processing systems. Through the use of an actual case study this paper aims to demonstrate that the use of formal specification techniques to develop a specification for a typical commercial data processing system is both practical and beneficial.

INTRODUCTION

Over the last few years the interest in formal specification techniques has increased dramatically and more and more emphasis has been placed upon the quality of software systems. It has been well documented that formal specifications can be of great benefit in the development of high quality software (Cohen [1]) and many different forms of specification have evolved, notably **Z**, VDM, CSP and Raise. It is common place to find at least one of these specification techniques being taught on Computer Science degree courses. However, most of the published examples of formal specifications tend to be for systems software or safety critical systems. In texts, some commercial data processing systems are used as illustrations but these are often small scale or greatly simplified and often do not address some common commercial data processing issues. (Diller [2], Potter et al [3]) This paper

546 Software Quality Management

presents how a formal specification using **Z** for an actual commercial data processing system can be developed, and will highlight the benefits gained. The paper also draws attention to the potential problems in using formal specifications and proposes solutions to overcome them.

THE CASE STUDY

The case study used in this paper is a system that was specified, designed and implemented for a property management company in Hong Kong. This company which is similar in nature to an estate agent, sells, and manages the rental of, residential properties and requested a computer system to replace their existing manual method of operation.

The choice of **Z** (Spivey [4], [5]) was made since this is the formal specification language taught to students at City Polytechnic and this exercise could then be used as a detailed case study on those courses. In addition, **Z** appears to be more modular than some of the other formal specification techniques and this would lead to an ideal architecture for the system when implemented.

User requirements

The user requirements for the system were given as follows:-

- To store and maintain details of all properties that are managed by the company. This is to include all properties that are for sale, for rent, or either for sale or rent.
- To store and maintain details of the owners of the properties referred to above.
- To store and maintain details of customers who are interested in buying or renting a property.
- To store and maintain details of customers who are presently renting properties.
- To allow enquires on the availability of properties that are for sale, for rent, or both based upon the price, size, or district. Enquires should also be possible for any combination of these attributes.
- Each property is to be assigned a unique identification code.
- Each customer is to be assigned a unique customer code.
- The properties are to be stored in identification code sequence.
- Each owner is to be assigned a unique identification code.

- Monthly lists detailing the current properties available for sale, and for rent are to be produced. These lists will be in descending order of price.

THE SPECIFICATION

The fundamental purpose of a specification is to model the *behaviour* of the system without detailing the structure of any records, files or database. This specification should be concise and unambiguous. The many traditional specification techniques for commercial data processing systems, for example, SSADM, DFD, Entity-Relationship modeling, whilst being concise, are not always unambiguous. This has led to systems being constructed which do not match their specifications and once a system is implemented it is very costly to correct errors and omissions. A formal specification, being mathematically based, is concise and not ambiguous and therefore systems implemented from them should conform to their specifications. In order to construct a formal specification it is necessary to have a thorough understanding of the system and this has the advantage of forcing an extensive analysis to be conducted. Formal specifications are not intended to be a replacement for the more common techniques of specification, rather, they are intended to be used as “front-end” to these techniques providing for them, a sound and accurate foundation from which to build upon. This enhances the overall quality of the system specification.

The steps involved in using **Z** in developing a formal specification (Bowen [6]) are:-

1. identify the basic types (sets).
2. specify the state of the system and any global operations that may be required. This models the associations between the basic types and states and any conditions that must always hold. These are known as the *state invariants*.
3. specify the operations, (identified from the user's requirements), that can be performed upon the system state. In this stage several iterations are usually necessary and probably involve modifications to the system state. **Z** encourages a modular approach and allows operations to be combined in various ways to form other operations.
4. prove various statements about the system's behaviour. Contrary to popular belief this not a compulsory part of a formal specification, but can be a useful aid to ensuring the correctness, and quality of a specification.

548 Software Quality Management

The basic types

Using the **Z** notation, the basic types used are as follows:-

[*PROPERTY*, *CUSTOMER*, *OWNER*, *STATUS*, *SALEPRICE*,
RENTPRICE, *SIZE*, *DISTRICT*, *PROPID*, *CUSTID*, *OWNERID*,
REPORT]

Where:-

PROPERTY is the set of all properties

CUSTOMER is the set of all customers

OWNER is the set of all owners

STATUS indicates whether a property is for sale only, for rent only either for sale or for rent or currently rented out

SALEPRICE is the set of all valid sale prices

RENTPRICE is the set of all valid rental prices

SIZE is the set of all valid property sizes

DISTRICT is the set of all valid districts

PROPID is the set of all unique property identification codes

CUSTID is the set of unique customer identification codes

OWNERID is the set of all unique owner identification codes

REPORT is the set of all messages, this type is required in all specifications

Constraints on the basic types

Unless otherwise specified, all basic types that have been introduced are infinite sets where the format of each element is not restricted. Therefore, explicit data structures are not part of the **Z** specification. However, certain basic types will need to be restricted both in terms of set size and in format of elements. This is as shown below:-

STATUS == { "For sale", "For rent", "Either", "Rented out" }

SALEPRICE == { 0.75 .. 30.0 }

RENTPRICE == { 7000 .. 85000 }

SIZE == { 600 .. 5000 }

DISTRICT == { "Clearwater bay", "Silverstrand", "Sai Kung",
"Ho Chung", "Hebe Haven", "Fei Ngo Shan",
"Kowloon" }

Note that there are two separate price ranges, (in fact they use different units) a fact easily specified but easily omitted from other less formal specification techniques. If the specification needs modification either during its development, or later during system enhancement, a single change in this part effects the change throughout the remainder of the specification, thus reducing the chance of errors.

Global operations — Sorting

In a specification, particularly one for a commercial system, sorting operations are commonplace. It is beneficial to make these operations global so that they can be reused later when needed. In this case study it can be concluded that three sort operations are needed, ascending property identification codes, descending sale price and descending rental price.

The formal specification of sorting depends upon there being a total order (Diller [2], Potter et al [3]) available for the basic type being operated upon. This corresponds to, in programming terms, the ability to be able to compare two instances of the same data object with respect to their magnitude. As property identification codes are alphanumeric and the prices numeric, there is no difficulty in stating that the total orders exist.

Specification of total order The specification of a total order is generic so as to be available for any defined type. It is specified as follows:-

$[X]$
$totorder : \mathbb{P}(X \leftrightarrow X)$
$totorder = \{ _ \sqsubseteq _ : X \leftrightarrow X \mid$
$\quad \forall x, y, z : X \bullet$
$\quad (x \sqsubseteq x) \wedge$
$\quad (x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y) \wedge$
$\quad (x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z) \wedge$
$\quad (x \neq y \Rightarrow x \sqsubseteq y \vee y \sqsubseteq x) \}$

Specification of sort The above specification can then be used in another generic schema to specify a general sorting procedure:-

$[X]$
$sortinorder : (\mathbb{F} X \times totorder[X]) \rightarrow seq X$
$sortinorder =$
$\{ nset : \mathbb{F} X; order : totorder[X]; nseq : seq X \mid$
$\#nseq = \#nset \wedge nset = ran nseq \wedge$
$(\forall i, j \in 1.. \#nset \mid i < j \bullet (nseq\ i, nseq\ j) \in order)$
$\bullet (nset, order) \mapsto nseq \}$

Note that this operation will be true for both ascending and descending sorts the distinction being made in the contents of the total order relation. These two generic schema operations are standard for specifications using sorts and therefore could be re-used in future specifications.

Specification of the global sorting operations These operations can now be used in the specification of the sorts needed for this system:-

550 Software Quality Management

$propertyidorder : totorder[PROPID]$	$[ascending]$
$salepriceorder : totorder[SALEPRICE]$	$[descending]$
$rentpriceorder : totorder[RENTPRICE]$	$[descending]$

$$sortpropertyid : \mathbb{F} PROPID \rightarrow seq PROPID$$
$$\forall p: \mathbb{F} PROPID \bullet$$
$$sortpropertyid p = sortinorder(p, propertyidorder)$$
$$sortsaleprice : \mathbb{F} SALEPRICE \rightarrow seq SALEPRICE$$
$$\forall p: \mathbb{F} SALEPRICE \bullet$$
$$sortsaleprice p = sortinorder(p, salepriceorder)$$
$$sortrentprice : \mathbb{F} RENTPRICE \rightarrow seq RENTPRICE$$
$$\forall p: \mathbb{F} RENTPRICE \bullet$$
$$sortrentprice p = sortinorder(p, rentpriceorder)$$

It should be noted that since the input for the above is a finite set, duplicate items are not allowed and would not appear in the final output. If duplicates were required in sorting operations, then the above can be respecified using a sequence as the input instead of the finite set.

Specification of the system state

In this section the relationships between the basic types are specified. Such questions as *How many properties can an owner own?* are posed and answered. This forces a thorough review and examination of the user requirements. The uniqueness of identification codes are also specified in this section along with any state invariants. Examination of the user requirements in the case study yielded the following additional facts to be specified in the system state:-

- all properties are considered to be available to customers, except those which have been rented out
- each property must have at least one owner and an owner can own several properties
- customers who rent properties must be known to the system
- each property must have a status, price (selling, renting or both) a size and a district
- several properties may have the same price, size and be in the same district

PropertyDB

propertyids : seq *PROPID*
allproperties : *PROPID* $\leftrightarrow$ *PROPERTY*
available : *PROPID* $\leftrightarrow$ *PROPERTY*
rented : *PROPID* $\leftrightarrow$ *PROPERTY*
customers : *CUSTID* $\leftrightarrow$ *CUSTOMER*
owners : *OWNERID* $\leftrightarrow$ *OWNERS*
ownedby : *PROPID* $\leftrightarrow$ *OWNERID*
renters : *PROPID* $\leftrightarrow$ *CUSTID*
propertystatus : *PROPID* $\leftrightarrow$ *STATUS*
saleprice : *PROPID* $\leftrightarrow$ *SALEPRICE*
rentprice : *PROPID* $\leftrightarrow$ *RENTPRICE*
sizes : *PROPID* $\leftrightarrow$ *SIZE*
districts : *PROPID* $\leftrightarrow$ *DISTRICT*

$available \cap rented = \emptyset$
 $allproperties = available \cup rented$
 $dom\ allproperties = ran\ propertyids$
 $dom\ available \subseteq ran\ propertyids$
 $dom\ rented \subseteq ran\ propertyids$
 $dom\ ownedby = ran\ propertyids$
 $dom\ renters \subseteq ran\ propertyids$
 $dom\ propertystatus = ran\ propertyids$
 $dom\ saleprice \subseteq ran\ propertyids$
 $dom\ rentprice \subseteq ran\ propertyids$
 $dom\ sizes = ran\ propertyids$
 $dom\ districts = ran\ propertyids$
 $dom\ owners = ran\ ownedby$
 $ran\ renters \subseteq dom\ customers$

The use of the partial function ($\leftrightarrow$) in the above corresponds to the data relationship one-to-many, and the use of the relation ($\leftrightarrow$) corresponds to the data relationship many-to-many. These relationships are commonplace in commercial data processing systems, and so these constructs in **Z** are very convenient for modeling. The use of a sequence (seq) for the property identification codes is necessary since a sorted order can only be specified on sequences.

The Δ , Ξ and Initial states

When using **Z** operations are considered to either change or not change the system state. To enable this to be easily specified, **Z** uses a concept known as *before* and *after* states. These are combined into two predefined combinations called the Δ state which used to specify operations which change the system state, and the Ξ state which is used to specify operations which do not change the system state. The Initial state specifies the system

starting position and the proof of the existence of this state is obligatory.

The Δ and Ξ states Although the Δ and Ξ states are defined, it is usual to state the following, so that it is clear that the specification uses the standard meanings of these states.

$$\Delta PropertyDB \cong PropertyDB \wedge PropertyDB'$$

$$\begin{aligned} \Xi PropertyDB \cong \Delta PropertyDB \mid \\ &propertyids' = propertyids \wedge \\ &allproperties' = allproperties \wedge \\ &available' = available \wedge \\ &rented' = rented \wedge \\ &customers' = customers \wedge \\ &owners' = owners \wedge \\ &ownedby' = ownedby \wedge \\ &renters' = renters \wedge \\ &propertystatus' = propertystatus \wedge \\ &saleprice' = saleprice \wedge \\ &rentprice' = rentprice \wedge \\ &sizes' = sizes \end{aligned}$$

The initial state In most cases the initial state specifies that the system will start with all sets in the system state equal to their respective empty sets. However, in commercial systems, the system being specified may begin by importing an existing data set. This situation can be specified, by introducing a state variable to represent the imported data set and then specifying the equality (representing the import action) within the initial state. As the case study represents a brand new system, the initial state is simply:-

$$\begin{aligned} InitPropertyDB' \cong PropertyDB' \mid \\ &propertyids' = \emptyset \wedge allproperties' = \emptyset \wedge \\ &available' = \emptyset \wedge rented' = \emptyset \wedge \\ &customers' = \emptyset \wedge owners' = \emptyset \wedge \\ &ownedby' = \emptyset \wedge renters' = \emptyset \wedge \\ &propertystatus' = \emptyset \wedge saleprice' = \emptyset \wedge \\ &rentprice' = \emptyset \wedge sizes' = \emptyset \end{aligned}$$

Proof of existence This proof is informal, and is quite acceptable for most initial states. Since the emptyset is defined for sets of all types, all sets defined in the system state must exist. As all sets exist and all invariants are satisfied, the initial state must exist. For a more formal treatment refer to (Diller [2], Potter et al [3])

Specifying the operations

Having specified the system state and proving that the initial state exists, the operations to manipulate the system state can now be specified. All operations should indicate whether they succeed or fail and to achieve this all specifications need an operation something like the following:-

Success

rep! : *REPORT*

rep! = "Success"

The operations needed for the case study are extracted from the user requirements and can be classified into the following categories:-

- **maintenance** — Operations which add, change and delete new properties, customers and owners
- **change** — Operations which change the status of a property, like renting, selling and end of lease
- **enquiries** — Operations that retrieve a set of property identification codes corresponding to a set of input criteria. Such operations are, properties for sale, properties for rent, properties within a price range, properties within a size range and properties within a set of district values
- **reporting** — Operations to produce the current properties for sale and for rent

Since the specification of operations within these categories is somewhat similar, this paper will present only those examples which illustrate distinctive techniques.

Specifying the addition of new properties This is not so straightforward as adding a new property may, or may not involve the addition of a new owner, a fact which could be easily overlooked in less formal specification methods. In addition, when adding a new property a decision has to be made as to the manner in which property identification codes are generated. Either the user creates the new code and enters it, or, as in this case study, the system generates a new code internally. This gives the following two operations:-

NewPropertyNewOwner Δ PropertyDB $pd? : PROPERTY; od? : OWNERS; sp? : SALEPRICE$ $rp? : RENTPRICE; ps? : STATUS; size? : SIZE$ $d? : DISTRICT$ $ps? \in \{ \text{"For sale"}, \text{"For rent"}, \text{"Either"} \}$ $\exists pi : PROPERTYID; oi : OWNERID \mid$ $pr \notin \text{ran propertyids} \wedge oi \notin \text{dom owners} \bullet$ $\text{allproperties}' = \text{allproperties} \oplus \{pi \mapsto pd?\} \wedge$ $\text{available}' = \text{available} \oplus \{pi \mapsto pd?\} \wedge$ $\text{owners}' = \text{owners} \oplus \{oi \mapsto od?\} \wedge$ $\text{ownedby}' = \text{ownedby} \oplus \{pi \mapsto oi\} \wedge$ $ps? \in \{ \text{"For sale"}, \text{"Either"} \}$ $\Rightarrow (\text{saleprice}' = \text{saleprice} \cup \{pi \mapsto sp?\} \wedge$ $ps? \in \{ \text{"For rent"}, \text{"Either"} \}$ $\Rightarrow (\text{rentprice}' = \text{rentprice} \cup \{pi \mapsto rp?\} \wedge$ $\text{sizes}' = \text{sizes} \cup \{pi \mapsto size?\} \wedge$ $\text{districts}' = \text{districts} \cup \{pi \mapsto d?\} \wedge$ $\text{propertyids}' = \text{sortpropertyid}(\text{ran}(\text{propertyid} \frown < pi >))$ $\text{rented}' = \text{rented} \wedge \text{customers}' = \text{customers} \wedge \text{renters}' = \text{renters}$ NewPropertyOldOwner Δ PropertyDB $pd? : PROPERTY; oi? : OWNERID; sp? : SALEPRICE$ $rp? : RENTPRICE; ps? : STATUS; size? : SIZE$ $d? : DISTRICT$ $ps? \in \{ \text{"For sale"}, \text{"For rent"}, \text{"Either"} \}$ $oi? \in \text{dom OWNERS}$ $\exists pi : PROPERTYID \mid$ $pr \notin \text{ran propertyids} \bullet$ $\text{allproperties}' = \text{allproperties} \oplus \{pi \mapsto pd?\} \wedge$ $\text{available}' = \text{available} \oplus \{pi \mapsto pd?\} \wedge$ $\text{ownedby}' = \text{ownedby} \oplus \{pi \mapsto oi?\} \wedge$ $ps? \in \{ \text{"For sale"}, \text{"Either"} \}$ $\Rightarrow (\text{saleprice}' = \text{saleprice} \cup \{pi \mapsto sp?\} \wedge$ $ps? \in \{ \text{"For rent"}, \text{"Either"} \}$ $\Rightarrow (\text{rentprice}' = \text{rentprice} \cup \{pi \mapsto rp?\} \wedge$ $\text{sizes}' = \text{sizes} \cup \{pi \mapsto size?\} \wedge$ $\text{districts}' = \text{districts} \cup \{pi \mapsto d?\} \wedge$ $\text{propertyids}' = \text{sortpropertyid}(\text{ran}(\text{propertyid} \frown < pi >))$ $\text{rented}' = \text{rented} \wedge \text{customers}' = \text{customers} \wedge \text{renters}' = \text{renters}$ $\text{owners}' = \text{owners}$

Note that when specifying operations it is important to specify what does not change as well as that which does. In these operations there is one common pre-condition

$$ps? \in \{ \text{"For sale"}, \text{"For rent"}, \text{"Either"} \}$$

the behaviour of the system when this pre-condition is false also needs to be specified.

BadStatus

$\exists \text{PropertyDB}$

$ps? : \text{STATUS}$

$rep! : \text{REPORT}$

$ps? \notin \{ \text{"For sale"}, \text{"For rent"}, \text{"Either"} \}$

$rep! = \text{"Status incorrect"}$

A similiar schema *NoSuchOwner* to handle the situtaion when the pre-condition $oi? \in \text{dom OWNERS}$ in the schema *NewPropertyOldOwner*, is also specified. Now, the whole operation can be simply stated by combining the schemas *NewPropertyNewOwner*, *NewPropertyOldOwner*, *Success*, *BadStatus* and *NoSuchOwner* together:-

$$\begin{aligned} \text{DoAddNewProperty} \triangleq & ((\text{NewPropertyNewOwner} \wedge \text{Success}) \vee \text{BadStatus}) \\ & \vee \\ & ((\text{NewPropertyOldOwner} \wedge \text{Success}) \vee \text{BadStatus} \\ & \vee \text{NoSuchOwner}) \end{aligned}$$

This pattern of developing an operation is common to all subsequent operations which have pre-conditions.

Specifying the deletion of a property This operation is distinct from selling or renting a property since it models the operation of removing a property which is no longer managed by the company. In specifications of commercial data processing systems it is important to maintain the integrity of the system and to ensure that all system invariants will be true. This will involve removing the property identification code from all sets in which it could possibly appear.

DeleteProperty

$\Delta \text{PropertyDB}$

$pd? : \text{PROPID}$

$$\begin{aligned} &pd? \in allproperties \\ &allproperties' = allproperties \setminus \{pd? \mapsto allproperties\ pd?\} \\ &ownedby' = ownedby \setminus \{pd? \mapsto ownedby\ pd?\} \\ &sizes' = sizes \setminus \{pd? \mapsto sizes\ pd?\} \\ &districts' = districts \setminus \{pd? \mapsto districts\ pd?\} \\ &propertystatus' = propertystatus \setminus \{pd? \mapsto propertystatus\ pd?\} \\ &pd? \in \text{dom } available \Rightarrow available' = available \setminus \{pd? \mapsto available\ pd?\} \\ &pd? \in \text{dom } rented \Rightarrow rented' = rented \setminus \{pd? \mapsto rented\ pd?\} \\ &pd? \in \text{dom } renters \Rightarrow renters' = renters' \setminus \{pd? \mapsto renters\ pd?\} \\ &pd? \in \text{dom } saleprice \Rightarrow saleprice' = saleprice \setminus \{pd? \mapsto saleprice\ pd?\} \\ &pd? \in \text{dom } rentprice \Rightarrow rentprice' = rentprice \setminus \{pd? \mapsto rentprice\ pd?\} \\ &\#(ownedby \triangleright \{ownedby\ pd?\}) = 1 \\ &\quad \Rightarrow owners' = owners \setminus \{ownedby\ pd? \mapsto owners\ ownedby\ pd?\} \quad [1.1] \\ &\#(ownedby \triangleright \{ownedby\ pd?\}) > 1 \\ &\quad \Rightarrow owners' = owners \quad [1.2] \\ &propertyids' = \quad [2] \\ &\quad sortedpropertyid(squash(propertyids \setminus \{(propertyids^{-1}\ pd?) \mapsto pd?\})) \\ &customers' = customers \end{aligned}$$

The labelled predicates within this schema are worthy of further comment. The lines labelled [1.1] and [1.2] specify that if the owner of the property being deleted does not own any more properties, then the details of that owner should be removed from the system. Otherwise the owner details are left unchanged. The line labelled [2] specifies that the property identification codes remain sorted after the removal of the property. As usual an operation is specified to satisfy the event of the pre-condition being false and then combined to form the overall operation:-

$$DoDeleteProperty \hat{=} (DeleteProperty \wedge Success) \vee NoSuchProperty$$

Specification of renting a property This specification is straightforward and has been included to illustrate that not not all schema defined operations are as complex as the two presented so far. Essentially, this is just a matter of changing an entry in one set, removing an entry from the *available* set and then adding this entry to the set *rented*. A new entry is added to the set *renters* to keep information about the customer renting the property.

RentAProperty

$\Delta PropertyDB$

$c? : CUSTID$

$p? : PROPID$

$c? \in \text{dom customers}$

$p? \in \text{dom available}$

$propertystatus\ p? \in \{ \text{"For rent"}, \text{"Either"} \}$

$available' = available \setminus \{p? \mapsto available\ p?\}$

$rented' = rented \cup \{p? \mapsto available\ p?\}$

$propertystatus' = propertystatus \oplus \{p? \mapsto \text{"Rented out"}\}$

$renters' = renters \cup \{p? \mapsto c?\}$

$propertyids' = propertyids \wedge allproperties' = allproperties$

$owners' = owners \wedge ownedby' = ownedby \wedge saleprice' = saleprice$

$rentprice' = rentprice \wedge sizes' = sizes \wedge districts' = districts$

In this operation there are three separate pre-conditions:-

$c? \in \text{dom customers}$

$p? \in \text{dom available}$

$propertystatus\ p? \in \{ \text{"For rent"}, \text{"Either"} \}$

As before operations are specified to handle the condition when each of these pre-conditions are false, (one of these operations can be reused from the specification of deleting a property) and these are then combined with the above to form the overall operation:-

$$DoRentProperty \triangleq (RentProperty \wedge Success) \vee UnknownCustomer \vee NoSuchProperty \vee PropertyNotForSale$$

Specification of enquiry — Properties for sale All the enquiry operations are specified in a similar manner and involve a two stage process. First obtain the property identification codes which corresspond to the enquiry criteria, then use this set to obtain the related property details. As the latter operation is common to all enquiries, it can be specified first and then reused when needed.

GetPropertyDetails

$\exists PropertyDB$

$propids? : \mathbb{P} PROPID$

$props! : \mathbb{P} PROPERTY$

$props! = \text{ran}(propids? \circ available)$

Now to specify the enquiry for properties that are for sale.

558 Software Quality Management

PropertiesForSale

 $\exists \text{PropertyDB}$
 $\text{minprice?}, \text{maxprice?} : \text{SALEPRICE}$
 $\text{forsale!} : \mathbb{P} \text{PROPID}$
 $\text{rep!} : \text{REPORT}$
 $\text{forsale!} = \text{saleprice}^{-1}(\{ \text{minprice} \dots \text{maxprice} \})$
 $\# \text{forsale!} > 0 \Rightarrow \text{rep!} = \text{"Success"}$
 $\# \text{forsale!} = 0 \Rightarrow \text{rep!} = \text{"No properties in this price range"}$

In this schema the error and success actions have been combined into the operation schema, to combine these two operations together, the output of the second is piped to the input of the first.

$$\text{DoPropertiesForSale} \hat{=} \text{PropertiesForSale} \gg \text{GetPropertyDetails}$$

Specification of the monthly for sale report This operation is very similar to the previous one, except that the output is to be in descending sale price order. Using the previously specified sort operation the specification becomes:-

ForSaleReport

 $\exists \text{PropertyDB}$
 $\text{forsale!} : \mathbb{P} \text{PROPID}$
 $\text{rep!} : \text{REPORT}$
 $\exists \text{sp} : \text{seq SALEPRICE} \bullet$
 $\text{sp} = \text{sortsaleprice}(\text{ran saleprice}) \wedge$
 $\text{forsale!} = \text{sp} \circ \text{saleprice}^{-1}$
 $\# \text{forsale!} > 0 \Rightarrow \text{rep!} = \text{"Success"}$
 $\# \text{forsale!} = 0 \Rightarrow \text{rep!} = \text{"No properties for sale"}$

$$\text{DoForSaleReport} \hat{=} \text{ForSaleReport} \gg \text{GetPropertyDetails}$$

BENEFITS, PROBLEMS & SOLUTIONS

Having presented the formal aspects of the specification the benefits, problems and suggested solutions can now be addressed.

Benefits from using formal specifications

The main overriding benefit in using formal specifications for commercial data processing systems is that the specification is unambiguous. The mathematical representation of the system can only be interpreted one way and

its operation is clearly defined. The integrity of the system is also clearly specified and all operations ensure that this integrity is maintained. The specification offers a concise, complete and correct platform from which the system can be implemented, this allows the implementor to concentrate on that issue and not be involved in specification issues. For each function the pre and post conditions are clearly defined and this allows the development of rigorous test cases to be developed easily. (Dick [7]) Another benefit, particularly in Hong Kong, is that the language of mathematics is international and therefore any language translation errors, that could occur with narrative specifications, are eliminated. As formal specifications are largely a “paper exercise” deriving a formal specification is cost effective and corrections are much cheaper than they would be during implementation.

Problems that occur when using formal specifications

The most serious problem when using formal specifications for commercial systems, is lack of knowledge. Most personnel in the data processing industry are not conversant with the technique, or the representation of, formal specifications. Most programmers would not be able to understand a formal specification and therefore could not implement a system from one. Although training courses do exist, training is a very expensive overhead and many organisations are not willing to pay the price for what they perceive to be of little benefit. There is an inherent conservatism amongst the data processing industry, with few willing to be the first to try something new, resulting in formal specification knowledge entering the industry slowly. In today's age of automation, analysts and programmers expect to have a variety of tools at their disposal. Although tools do exist for Z , notably CADiZ (Jordan et al [8]) and $fUZZ$ (Spivey [9]), both of these rely on a knowledge, and use of $LATEX$ (Lamport [10]) which is almost unknown in the commercial data processing industry, this adds to the lack of incentive to try formal specification techniques.

Suggested solutions

It is widely accepted that to fully engineer a product that product has to be specified exactly, and accurately. Usually this involves some form of mathematical model. It makes sense therefore, that in order to label ourselves “software engineers” we should also specify software systems exactly and accurately using mathematical models. To make formal specifications more attractive to the data processing industry, this paper proposes following the engineering industry more closely by creating a specialised department. This department, referred to as the “technical design centre”, would take the documents and reports produced by systems analysts, produce a formal specification (developed along the lines highlighted in this paper) and then convert this formal specification into a format suitable for implementation. This format could be a program structure chart, pseudocode or

even an object-oriented design. This approach would be cost effective since, minimal re-training would be required, staff for this new department would come from new graduates and existing professionals with detailed knowledge of formal specification techniques. To support this department, new tools will have to be developed, that allow for the creation and animation of formal specifications. Such a tool is currently being researched at City Polytechnic of Hong Kong.

CONCLUSION

This paper has aimed to illustrate that formal specification techniques can be applied to commercial data processing systems, and whilst some of the stages may be fairly complex, once specified they can be reused in future specifications. Formal specifications can be beneficial in the commercial data processing industry, but certain problems prevent their widespread use. This paper has proposed some solutions to these problems and illustrates that formal specification techniques could, one day, be effectively used in the specification of commercial data processing systems.

REFERENCES

- [1] B. Cohen. Justification of formal methods for system specifications & A rejustification of formal notations. *Software Engineering Journal* 4(1):26-38, January 1989
- [2] A. Diller. *Z: An Introduction To Formal Methods* Wiley, Chichester, UK, 1990
- [3] B.F. Potter, J.E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK 1990
- [4] J.M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1), January 1989
- [5] J.M. Spivey. *The Z Notation: A Reference Manual* International Series in Computer Science, Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2nd Edition, 1992
- [6] J.P. Bowen. Formal specification in Z as a design and documentation tool. In *Proc. Second IEE/BSC Conference on Software Engineering*, volume 290, pages 164-168. IEE/BSC, July 1988
- [7] J. Dick & A. Faivre. Automating the generation and sequencing of test cases from model based specifications. In J.C.P. Woodcock & P.G.



Larsen, editors, *FME'93 Industrial Strength Formal Methods*, Lecture Notes in Computer Science 670, Springer-Verlag, pages 268-284, 1993

- [8] D. Jordan, J.A. McDermid, and I. Toyn. CADiZ - computer aided design in Z. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computinf, pages 93-104. Springer-Verlag, 1991
- [9] J.M. Spivey. *The fUZZ Manual*. Computing Science Consultancy, 2 Willow Close, Garsington, Oxford, OX9 9AN, UK, 2nd Edition, 1992
- [10] L. Lamport. *L^AT_EX User's Guide & Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1986