Vertical Z

Technical Report No.132

Martin Loomes and Carol Britton

May 1991

Vertical Z

Martin Loomes and Carol Britton
School of Information Science,
Hatfield Polytechnic,
College Lane, Hatfield, Herts, AL10 9AB, UK2
tel: 0707 279350

Richard Mitchell
Faculty of Information Technology,
Brighton Polytechnic
Lewes Road, Brighton BN2 4GJ, UK.
tel: 0273 600900

Keywords: Formal, Specification, Decomposition

May 22, 1991

Introduction - Horizontal and Vertical Decomposition.

Many descriptions or definitions of systems and their sub-systems are sufficiently long that they need to be organised, to be given some internal structure. This internal structure is achieved by some form of decomposition of the whole into parts (together, of course, with the means to compose the parts). The specification language Z (Spivey 1989) provides the schema calculus as the means to decompose Z specifications. The nature of the schema calculus encourages specifiers to use what we are going to term horizontal decomposition. Briefly, this involves taking a description of some behaviour that involves a number of different cases, and presenting each case in a different schema. In contrast, other languages, notably programming languages such as Pascal, encourage what we are going to term vertical decompostion. In programming terms, this involves presenting an algorithm not as one monolithic piece of code but as a higher-level procedure defined in terms of lower-level procedures. (For a full description of vertical and horizontal decomposition see Mitchell, Loomes and Howse 1990)

Decomposition, whether vertical or horizontal, is a mechanism for structuring specifications. To use it, the specifier must have a strategy for choosing what goes into different components. The specification presented in this paper uses the strategy of separating two aspects of the system: one aspect is the essential functionality of the system; the other aspect is how this essential functionality is presented to the system's users. We agree with Meyer (1988) that "... healthy design methods will attempt, as much as possible, to separate the interface from the rest of the system, and use deeper properties as a guide to system structuring."

In this paper, lessons learned about decomposition in programming are used as the basis for the organisation of a Z specification. The example used is drawn from an information system which manages security for a large company. The part of the specification shown here illustrates the use of cards to permit access to secure rooms in the company building. We will refer to this part of the specification as the Door Control System.

The Door Control System

To produce a clear, well-structured specification we need to separate the essential functionality of the system (the checking of cards in and out of secure rooms) from its interface (the way in which it reacts with its users). This separation of concerns brings us two advantages: first,we will have a better understanding of the different components of the system; second, we have the ability to modify the interface at a later date without affecting the basic functionality of the system.

We start by defining the data types that we are going to use to capture the values used in the system, then we define the user interface and the effect of the externally visible operations in terms of changes to stored variables of the defined types. Finally we define a number of functions for manipulating these values.

The Data Model

To specify the Door Control system we need to use the types *Card* and *Room* We leave both the types free, as we are not bothered about the exact format of these values at present.

At any given moment the system will have to know certain facts;

- which cards are registered as part of the door control system
- which cards (and therefore employees) are allowed in which rooms
- which cards (employees) are at present in a secure room

To keep track of which cards are in which rooms we will use a partial function.

$$In == Card \pfun Room$$

To record who is allowed into which project rooms we will use another partial function, one that maps a card to a set of rooms that the cardholder is entitled to enter.

$$Permissions == Card \pfun \mathbb{P}\, Room$$

We will eventually require a variable to hold all this information, so we define a type corresponding to values which capture both the record of which card is in which room, and also what permissions all cardholders employees have. This is simply a tuple of the two functions described above.

$$State == In \times Permissions$$

Some of our system operations will only change one part of this state, so it is convenient to have functions that allow us to extract just the first or second component of the state tuple. We will call these two functions *inComponent* and *permissionComponent* respectively. These are defined as follows:

STATE_TYPE_SELECTORS

$$inComponent : State \to In$$
$$permissionComponent : State \to Permissions$$

$$inComponent(i, p) = i$$
$$permissionComponent(i, p) = p$$

The State Variable

We are now going to pave the way for describing our externally visible operations. To do this we need to describe how the state actually changes as we perform the operations, and any responses the system makes. In this schema we also introduce an invariant on the state. We will insist that cardholders are not allowed to be in rooms that they do not have permissions to be in.

ΔSTATE_VARIABLE

$$STATE\_TYPE\_SELECTORS$$
$$st, st' : State$$

$$\forall c : Card \bullet (inComponent\ st\ c) \in (permissionComponent\ st\ c)$$
$$\land\ (inComponent\ st'\ c) \in (permissionComponent\ st'\ c)$$

The Outer Layer - The System Interface

We can now define our user interface. This involves specifying the behaviour of the system, in terms of outputs and changes of state, corresponding to the invocation of operations with particular inputs. In this paper we will use the operation to allow or deny entry as an example. We will first provide a case analysis that describes the behaviour of the operation under all possible situations. We will also provide a response to the user, so that acknowledgements for actions, or messages saying why the operation won't behave the way the user expected, can be provided. Finally we will formalize the case analysis in a Z schema.

Entering a Room

When an employee attempts to enter a room, several things may go wrong. First, the card being used may not be recognised by the system. Second, the card may not have permission for the room being entered. Finally, the employee may already be in a room (as far as the system knows). This situation might arise if an employee passes a card out through a window to an accomplice, or if an employee sneaks out through a door without using a card (eg. with someone else).

We assume that the output of a response "Door unlocked" here is accompanied by the physical unlocking of the door, and other messages may be accompanied by alarm bells, messages to the security controller, or whatever is deemed appropriate.

CASE 1 Card not known to the system
Output the mesage "Card not known" and do nothing.

CASE 2 Card is known to the system

CASE 2.1 The card does not have permission for the room being entered.
Output "Permission denied", and do nothing

CASE 2.2 Permission is held for the room being entered.

CASE 2.2.1 The card is already logged as being in a room
Output "Card is already in secure room" and do nothing

CASE 2.2.2 The card is not logged as already in a room
Output "Door unlocked" and admit the employee.

This can be formalised in the following schema:

ENTRY_REQUEST

$\Delta STATE\_VARIABLE$
$ENTER\_ROOM\_FUNCTION$
$r? : Room$
$c? : Card$
$resp! : Response$

$$
\begin{array}{l}
(c? \notin \mathrm{dom}(permissionComponent\ st) \\
\quad \wedge \\
\quad resp! = \text{"}Card\ not\ known\text{"} \\
\quad \wedge \\
\quad st' = st) \\
\vee \\
(c? \in \mathrm{dom}(permissionComponent\ st) \\
\quad \wedge \\
\qquad (r? \notin (permissionComponent\ st\ c?) \\
\qquad \wedge \\
\qquad resp! = \text{"}Permission\ denied\text{"} \\
\qquad \wedge \\
\qquad st' = st) \\
\qquad \vee \\
\qquad (r? \in (permissionComponent\ st\ c?) \\
\qquad \wedge \\
\qquad\quad (c? \in \mathrm{dom}(inComponent\ st) \\
\qquad\quad \wedge \\
\qquad\quad resp! = \text{"}Card\ is\ already\ in\ secure\ room\text{"} \\
\qquad\quad \wedge \\
\qquad\quad st' = st) \\
\qquad\quad \vee \\
\qquad\quad (c? \notin \mathrm{dom}(inComponent\ st) \\
\qquad\quad \wedge \\
\qquad\quad resp! = \text{"}Door\ unlocked\text{"} \\
\qquad\quad \wedge \\
\qquad\quad st' = enterRoom(st, c?, r?))
\end{array}
$$

At this stage we have not yet defined the function *enterRoom*. This is because we are concentrating here on the outer layer of the system - its interface with its users and the way it reacts under various conditions. For the moment all we need be concerned about is that a successful attempt to enter a room brings about

a new system state. This new state is the result of applying a function which we have called *enterRoom* to the old system state,a card and a room. We can leave consideration of how *enterRoom* actually works until later in the specification.

The Inner Layer - Functions to Manipulate the State Variable

Now that we have specified the ways in which the system will react to various situations, we need to define a number of functions to manipulate values of type *State*. These functions should not be confused with the *operations* that we have already defined to provide our user interface. Our definition of this interface prescribes under what conditions the functions can be invoked. Because we have already specified this, we do not need to worry about imposing preconditions on our functions at this stage. All of the functions make sense without preconditions, it is only that their *use* may be inappropriate in certain situations. This technique of separating the functions from the different ways in which they may be used will allow maximal re-use of the functions at a later stage.

We will define the function *enterRoom* which is called in the Entry Request Operation specified above. This function calculates new values of type *State* that result from entry to a room.

ENTER_ROOM_FUNCTION

$$STATE\_TYPE\_SELECTORS$$

$$enterRoom : State \times Card \times Room \rightarrow State$$

$$enterRoom(st, c, r) =$$
$$(inComponent\ st \oplus \{c \mapsto r\}, permissionComponent\ st)$$

What have we achieved so far?

The door control specification was developed in two distinct parts - the outer layer or interface of the system, and the inner layer or system functionality. This separation has several advanatges. It breaks down the problem area into smaller, more manageable sections, thus giving us a more thorough understanding of the system during the development of the specification. Any problem has to be decomposed into 'brain-sized chunks' before we can start to tackle it. Disconnecting the interface from the functionality of the system is a useful and effective way of achieving this problem decomposition. The separation of the specification into distinct components simplifies modification of it at later stages, since we can alter the way in which we have defined either the functions or the operations on the system without affecting the other component. As development of a system progresses modification is always necessary. Good decomposition of the problem means that we can avoid massive rewrites to accommodate small changes and reuse the parts of our specification which are still relevant.

References

Meyer, Bertrand, "Object-oriented Software Construction", Prentice-Hall, 1988, p46

Mitchell, R, Loomes, M and Howse, J, "Organising Specifications: a Case Study", 1990

Spivey, J M, "An Introduction to Z and Formal Specifications", IEE Software Engineering Journal, January 1989