Structuring Z Specifications with Views

Daniel Jackson
June 1995
CMU-CS-94-126R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A view is a partial specification of a program, consisting of a state space and a set of operations. A full specification is obtained by composing several views, linking them through their states (by asserting invariants across views) and through their operations (by defining external operations as combinations of operations from different views).

By encouraging multiple representations of the program's state, view structuring lends clarity and terseness to the specification of operations. And by separating different aspects of functionality, it brings modularity at the grossest level of organization, so that specifications can accommodate change more gracefully.

View structuring in Z is demonstrated with a few small examples. Both the features of Z that lend themselves to view structuring, and those that are a hindrance, are discussed.

This research was sponsored in part by a Research Initiation Award from the National Science Foundation (NSF), under grant CCR-9308726, by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA), under grant F33615-93-1-1330. Views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of ARPA, NSF, TRW or the United States Government.

Keywords: formal specification, requirements, Z, views.

1 Introduction

The structure of most published Z specifications follows, quite closely, the structure of an implementation. At the lowest level, of course, the structures diverge, and predicates over sets replace loops, pointers and so on. But the gross organization often retains the flavour of a program, with global variables brought into a common area and operations packaged into modules. Even procedure call has its analogue – in promotion, a technique in which the local state of a schema is bound to a component of the global state, like the binding of formals to actuals.

Conjunction is the lynchpin of implicit specification, and brings its most significant benefit: separation of concerns. While the code of an operation must exhibit several properties at once (obeying the Shanley principle of traditional engineering), its specification may separate them. A line justification algorithm must find hyphenation points and distribute spaces to optimize layout, but its specification need only say that lines have fixed length, *and* hyphens are inserted according to the dictionary, *and* rivers of white space are absent. The advantages of this separation are clarity, terseness and modularity. The main disadvantage – the dark side of conjunction – is the risk of overconstraint: there may be no layout of the text that meets all the requirements. But in the early stages of development, the risk is worth taking (and can be alleviated to some degree by extra vigilance, in Z by calculating preconditions and in Larch and VDM by checking implementability).

Implicit specification, then, is a powerful technique commonly exploited at the level of operations but rarely at higher levels. This paper illustrates a structuring style in which implicit specification plays a role at the grossest level. The program is specified as the composition of several *views*. Like a module, a view defines a state and some operations. But views are composed more freely than modules: an operation may appear in more than one view, and the operations of the program as a whole may be formed by various combinations of view operations.

Views decouple the aspects of a program's functionality, so that each can be constructed (and embellished) independently. A specification of a word processor, for instance, might separate text-oriented functions, such as search/replace and checking spelling, from typographic functions, such as justification. Each view has its own representation of the state space, so the text-oriented functions might be defined over a string of alphabetic characters, while the typographic functions might call for a more elaborate state with soft-hyphens, ligatures, kerning, etc. Nasty questions about interaction between aspects (what happens if you replace a word that straddles a soft line break? when do letter pairs become ligatures?) can be postponed until the views are composed, and are more easily resolved than if the aspects were intertwined from the start.

View structuring is evident, to a greater or lesser extent, in many published Z specifications. Redundant state components are often declared purely to ease the definition of certain operations. Multiple representations have been used on a larger scale too, as in Sufrin's editor specification [Suf82], which relates the appearance of a text

buffer on the screen (given as a sequence of lines) to its internal representation (a sequence of characters).

In other languages, the kind of redundancy and pervasive use of invariants common to Z specifications is not encouraged and view structuring is thus rarely seen. In VDM [Jon90], invariants are associated with types rather than states, and as in Larch [GHW85], invariants on states are treated as proof obligations and are not part of the specification proper.

This paper contains no radical novelties. Its intent is to articulate, by means of small illustrations in standard Z [Spi92], a style of specification based on views. It also attempts to explain why Z is especially well suited to view structuring, pointing to features that have not been stressed in recent comparisons [Hay92, HJN93, Hal93], and also to note some deficiencies of Z that make view structuring less natural than it might otherwise be.

2 Why Views?

Views respond to a simple dilemma. The first step in writing a conventional model-based specification is to define the state space. How the states are represented largely determines how easy it is to define the operations, so finding a good representation can be hard. Sometimes no single representation does the trick; some operations call for one, and some another.

Take cursor motion in an editor buffer, for example. A nice representation [Suf82] of a buffer is two sequences of characters, one for the text preceding the cursor and the other for the text following it:

File

$$left, right : \mathrm{seq}\ Char$$

Moving the cursor to the left transfers a character from left to right:

csrLeft

$$\Delta File$$

$$\exists c : Char \bullet left = left' \frown \langle c \rangle \land right' = \langle c \rangle \frown right$$

and inserting a character extends the sequence on the left:

insertChar

$$\Delta File$$
$$c : Char$$

$$left' = left \frown \langle c \rangle \land right' = right$$

Now consider moving the cursor up one line. Thinking that this is equivalent to a series of leftward motions, we might look for an appropriate suffix of *left* to amputate

and append as a prefix to *right*. But this suffix is not easy to define. If the text is wrapped automatically, so that soft line breaks are inserted in the course of typing, the size of this suffix cannot be determined without knowing the position of the last line break in *left*, which in turn depends on the placement of word separators in *left*'s entirety!

A better state representation for this operation is a sequence of lines, where each line is itself a character sequence of some maximum length that ends in a carriage return or space and has no carriage returns anywhere else:

Grid

$$lines : seq\ seq\ Char$$
$$x, y : \mathbb{N}_1$$

$$y \in dom\ lines \land x \in dom\ lines(y)$$
$$\forall l \in ran\ lines \bullet \#l \leq max \land last(l) \in \{spc, cr\} \land cr \notin front(l)$$

The cursor is now represented as a pair of positive integers that index the character immediately following it. Moving the cursor up is now easy to define:

csrUp

$$\Delta\ Grid$$

$$y' = y - 1$$
$$x' = min\ \{x, \#lines(y')\}$$
$$lines' = lines$$

as are other line-based operations that are tricky to define on the first representation, such as deleting a line:

delLine

$$\Delta\ Grid$$

$$lines' = (1..y-1) \lhd lines \frown (y+1\ ..\ \#lines) \lhd lines$$
$$x' = 1 \land y' = y$$

This representation, however, is no good for the previous operations. The behaviour of *insertChar* depends on what character is inserted (since carriage returns break lines) and the length of the line (to preserve the bound). The specification of backwards deletion would likewise involve a clumsy splitting into cases (to handle the start of the line).

The solution to this dilemma is to divide the specification into *views*. Each view can have a different state representation, and an operation can be specified in one or more views. Here the character sequence operations fill one view (*File*) and the line operations another (*Grid*). An invariant between the states of the two views ensures that they match as expected. The views must agree on the textual content both of the entire buffer, and of the portion preceding the cursor:

Editor

File
Grid

$left \frown right = \frown/lines$
$left = (\frown/(1 \mathinner{..} y-1) \lhd lines) \frown (1 \mathinner{..} x-1) \lhd lines(y)$

The operations previously defined must now be extended to act on the combined state of the two views:

$Editor\_insertChar \mathrel{\widehat=} [\Delta Editor \mid insertChar]$
$Editor\_csrLeft \mathrel{\widehat=} [\Delta Editor \mid csrLeft]$
$Editor\_csrUp \mathrel{\widehat=} [\Delta Editor \mid csrUp]$
$Editor\_delLine \mathrel{\widehat=} [\Delta Editor \mid delLine]$

Superficially this resembles the standard composition of two modules. But there is a crucial difference. There are no frame conditions that hold the state of one view invariant when an operation from the other is executed; on the contrary, almost any change to one will affect the other.

Figure 1 brings the specification fragments together, prefixing the names of the operations to show which view they belong to. The reader who doubts the benefit of two representations might try to recast an operation from one view in the representation of the other. A more substantial benefit than ease of expression, however, is the modularity views provide. If we consider some natural extensions to our specification, we will find that most fit neatly in a single view.

When the cursor is moved up or down it may jump to the left if otherwise it would land beyond the line. In some editors the cursor remembers its previous position, and will move back to the right again when taken to a longer line. This is a desirable feature, since it results in more natural behaviour (moving up then immediately down, for instance, has no effect). How might we add it to our specification? The feature is about lines, so the first place to look is the *Grid* view. The state would be extended with the cursor's "memory", and the *csrUp* and *csrDown* operations amended accordingly. Left and right movements of the cursor must reset the memory, in addition to having their normal effect. But this does not imply a change to *csrLeft* and *csrRight* in the *File* view. Instead local specifications of the two operations would be added to the *Grid* view. The aspect of functionality to do with cursor memory is thus confined within the appropriate view; its effect on the overall behaviour would be obtained by defining the external *csrLeft* and *csrRight* operations as the conjunction of their partial specifications in the two views.

Even drastic changes can sometimes be confined within a view. To change from a fixed-width character display to a bit-mapped screen with proportional spacing would need a new *Grid* view, but the *File* view would remain unscathed. Major enhancements willl usually require a new view, however. A *Word* view for specifying spelling checks might represent the buffer as a sequence of words, abstracting away

$Char == spc \mid cr \mid a \mid b \mid \ldots$
$Line == seq\ Char$
$max: \mathbb{N}_1$

File

$left, right: \text{seq}\ Char$

File_csrLeft

$\Delta File$

$\exists c: Char \bullet left = left' \frown \langle c \rangle \ \wedge \ right' = \langle c \rangle \frown right$

File_insertChar

$\Delta File$
$c: Char$

$left' = left \frown \langle c \rangle \ \wedge \ right' = right$

Grid

$lines: \text{seq}\ Line$
$x, y: \mathbb{N}_1$

$y \in \text{dom}\ lines \ \wedge \ x \in \text{dom}\ lines(y)$
$\forall l \in ran\ lines \bullet \#l \leq max \wedge last\ (l) \in \{spc, cr\} \ \wedge \ cr \notin front(l)$

Grid_csrUp

$\Delta Grid$

$y' = y - 1$
$x' = min\ \{x, \#\ lines(y')\}$
$lines' = lines$

Grid_delLine

$\Delta Grid$

$lines' = (1 \mathinner{..} y-1) \lhd lines \frown (y+1 \mathinner{..} \#lines) \lhd lines$
$x' = 1 \ \wedge \ y' = y$

Editor

File
Grid

$left \frown right = \frown / lines$
$left = (\frown / (1 \mathinner{..} y-1) \lhd lines) \frown (1 \mathinner{..} x-1) \lhd lines(y)$

$Editor\_insertChar \triangleq [\Delta Editor \mid File\_insertChar]$
$Editor\_csrLeft \triangleq [\Delta Editor \mid File\_csrLeft]$
$Editor\_csrUp \triangleq [\Delta Editor \mid Grid\_csrUp]$
$Editor\_delLine \triangleq [\Delta Editor \mid Grid\_delLine]$

Figure 1: An outline of an editor specification in two views

distinctions between separators, and joining syllables separated by soft-hyphens. An *Outline* view might structure the buffer as a sequence of sections, or perhaps as a tree.

Some extensions will require an elaboration of the invariant relating the views. Strengthening the *Editor* schema with the addition of the formula

$\forall ls: seq\ Line \bullet left \frown right = \frown / ls \Rightarrow \#\ ls \geq \#\ lines$

for example, specifies an (unrealistically demanding) auto-wrapping policy: that the division into lines be the shortest possible.

3 Reasoning about Views

Separation into views can help not only in the construction of a specification but also in its analysis. So long as views are combined by conjunction, a safety property inferred from a single view will hold for the entire specification. From the *Grid_delLine* schema (Figure 1), for example, we can extract the precondition

$y < \#lines$

so deleting the last line of the buffer is not allowed.

Sometimes we will want to reason about several operations in a single view, and it will be inconvenient unless all the operations are defined there. For example, we may want to show that any invocation of *csrUp* is equivalent to a sequence of *csrLeft*'s. Rather than recasting one operation in the other view – which we avoided for good reason when constructing the specification – we can include a redundant specification that is only partial. Adding

File_csrUp

$\Delta File$

$\exists l: \text{seq } Char \bullet left = left' \frown l \ \wedge \ right' = l \frown right$

to the *File* view, for example, allows us to infer the required relationship between *csrUp* and *csrLeft*, even though it gives only the barest outline of the operation's behaviour: that the text is unchanged and the cursor moves backwards. When the views are composed, this schema will not appear in the definition of the system. Instead, we would record a proof obligation:

$Editor\_csrUp \vdash File\_csrUp.$

So although the reasoning cannot be confined to a single view, we can at least factor it into two steps, minimizing the involvement of the inter-view invariant.

4 Joining Views by their Operations

In the editor specification, the two views are joined by an invariant relating their states, and each operation of the program is an operation on one view or the other, but never both. Another way to join views is to synchronize their operations; in this case, the states of the views need not be related explicitly at all.

A telephone can be described as a simple machine whose state is one of

$Status ::= ringing \mid idle \mid waiting \mid connected \mid dialtone \mid busytone \mid ringtone$

Our first view associates states with phones by mapping directory numbers (some arbitrary set Id) to $Status$:

Phones

$$ps : Id \pfun Status$$

Initiating a phone call means lifting the handset when the phone is idle, and it leads to a dialtone:

Initiate

$$\Delta Phones$$
$$i : Id$$

$$ps(i) = idle \land ps'(i) = dialtone$$

In contrast, lifting the handset when the phone is ringing is an instance of

Answer

$$\Delta Phones$$
$$i : Id$$

$$ps(i) = ringing \land ps'(i) = connected$$

The separation of the same action (lifting the handset) into two different classes of operation is determined locally by the state of the phone when the action is performed; later we shall see how this kind of classification plays a pivotal role in the view composition.

Both operations concern a phone i, and determine the change in value of $ps(i)$, that phone's state. Neither constrains the state of other phones at all. By defining the schema

Frame

$$\Delta Phones$$

$$c : \mathbb{P}\, Id$$
$$\forall i : (\mathrm{dom}\, phones \setminus \{c\}) \bullet ps(i) = ps'(i)$$

$[Id]$

$Status ::= ringing \mid idle \mid waiting \mid connected \mid dialtone \mid busytone \mid ringtone$

$Phones \mathrel{\widehat=} [ps: Id \pfun Status]$

$Phones\_Initiate \mathrel{\widehat=} [\Delta Phones; i: Id \mid ps(i) = idle \land ps'(i) = dialtone]$
$Phones\_Ring \mathrel{\widehat=} [\Delta Phones; i: Id \mid ps(i) = idle \land ps'(i) = ringing]$
$Phones\_Answer \mathrel{\widehat=} [\Delta Phones; i: Id \mid ps(i) = ringing \land ps'(i) = connected]$
$Phones\_Dial \mathrel{\widehat=} [\Delta Phones; i, to: Id \mid ps(i) = dialtone \land ps'(i) = waiting]$
$Phones\_GetRing \mathrel{\widehat=} [\Delta Phones; i: Id \mid ps(i) = waiting \land ps'(i) = ringtone]$
$Phones\_GetBusy \mathrel{\widehat=} [\Delta Phones; i: Id \mid ps(i) = waiting \land ps'(i) = busytone]$
$Phones\_GetConn \mathrel{\widehat=} [\Delta Phones; i: Id \mid ps(i) = ringtone \land ps'(i) = connected]$
$Phones\_LoseConn \mathrel{\widehat=} [\Delta Phones; i: Id \mid ps(i) = connected \land ps'(i) = dialtone]$
$Phones\_Hangup \mathrel{\widehat=} [\Delta Phones; i: Id \mid ps(i) = connected \land ps'(i) = idle]$
$Phones\_Replace \mathrel{\widehat=} [\Delta Phones; i: Id \mid$
$\quad ps(i) \in \{dialtone, busytone, ringtone, waiting\} \land ps'(i) = idle]$

Phones_Frame

$\Delta Phones$
$c: \mathbb{P}\, Id$

$\forall i : (\mathrm{dom}\ phones \setminus \{c\}) \bullet ps(i) = ps'(i)$

Figure 2: The Phones *view*

saying that all phones except those in the set c maintain their state, we can form more restrictive operations. The conjunction

$Initiate \land (\exists c: \mathbb{P}\, Id \mid c = \{i\} \bullet Frame)$

for example, defines the action in which phone i initiates a connection and all other phones are unchanged.

The operations on a phone constitute one view of the system (Fig. 2). Note that dialling (naively modelled as an atomic operation) takes an argument – the number of the phone being called – but does not use it in the schema. The response to dialling is either *GetBusy* or *GetRing*, but which of the two occurs is not specified in this view.

The switch that handles connections between phones is specified in a second view (Fig. 3). Its state is a set of requested connections (a partial function, since a phone can only dial one number at a time) and a set of active connections (an injection, since a phone can only receive a call from one phone at a time):

Switch

$reqconns: Id \pfun Id$
$conns: Id \rightarrowtail Id$

$conns \subseteq reqconns$
$dom\ conns \cap ran\ conns = \varnothing$

The schema's predicates say that an active connection is considered requested until it terminates [Mor93], and that no phone can be both a calling and a called party at once.

In this view, the dialling of a number by a phone is an instance of the *Request* operation, which takes as arguments the number of the phone making the call (*from*) and the number dialled (*to*). The switch executes a *Connect* when a request can be fulfilled (the called phone is not busy), or a *Reject* otherwise. *Terminate* ends the call.

The system state is obtained, as before, by combining the states of the two views:

$Net \mathrel{\widehat=} Phones \land Switch$

This time, however, there is no inter-view invariant: the only connection between the views will be through their operations. System operations are now obtained not only by extending operations from the views over the system state, but by combining operations from different views. For example,

$$Net\_DialReq \mathrel{\widehat=} \Delta Net \land (\exists c: Id \mid c = \{from\} \bullet Frame) \land Switch\_Request \land Phones\_Dial[from/i]$$

says that the *DialReq* event of the system consists of a *Request* operation executed by the switch and a *Dial* operation executed at a particular phone. Z's syntax unfortunately hides the arguments of the operations, so the renaming calls for explanation. *Switch_Request* has two arguments, *from* and *to*, representing the number of the phone making the request and the number it wants to connect to. *Phones_Dial* has two arguments likewise, *i* for the number of the phone executing the operation, and *to* for the number dialled. The renaming links the views by making the number of the requesting phone in the switch the number of the phone at which the dial operation occurs. Finally, the frame condition says that only the phone with number *from* may suffer a state change.

Some system operations involve state changes at more than one phone. When one phone is answered, another becomes connected (and the switch registers the connection):

$$Net\_Conn \mathrel{\widehat=} \Delta Net \land (\exists c: Id \mid c = \{from, to\} \bullet Frame) \land Switch\_Connect \land Phones\_Answer[to/i] \land Phones\_GetConn[from/i]$$

Switch

$reqconns: Id \nrightarrow Id$
$conns: Id \rightarrowtail Id$

$conns \subseteq reqconns$
$dom\ conns \cap ran\ conns = \varnothing$

Switch_Request

$\Delta Switch$
$from, to: Id$

$reqconns' = reqconns \cup \{from \mapsto to\}$
$conns' = conns$

Switch_Try

$\Delta Switch$
$from, to: Id$

$from \mapsto to \in reqconns \ \wedge \ to \notin (\text{dom}\ conns \cup \text{ran}\ conns)$
$conns' = conns \wedge reqconns' = reqconns$

Switch_Connect

$\Delta Switch$
$from, to: Id$

$from \mapsto to \in reqconns \ \wedge \ to \notin (\text{dom}\ conns \cup \text{ran}\ conns)$
$conns' = conns \cup \{from \mapsto to\}$
$reqconns' = reqconns$

Switch_Reject

$\Delta Switch$
$from, to: Id$

$from \mapsto to \in reqconns \ \wedge \ to \in (\text{dom}\ conns \cup \text{ran}\ conns)$
$reqconns' = reqconns \setminus \{from \mapsto to\}$

Switch_Terminate

$\Delta Switch$
$from, to: Id$

$from \mapsto to \in conns$
$conns' = conns \setminus \{from \mapsto to\}$
$reqconns' = reqconns \setminus \{from \mapsto to\}$

Figure 3: The Switch *view*

$Net \triangleq Phones \wedge Switch$

$Net\_Req \triangleq$
$\Delta Net \wedge (\exists c: Id \mid c = \{from\} \bullet Frame)$
$\wedge Switch\_Request \wedge Phones\_Dial[from/i]$

$Net\_Try \triangleq$
$\Delta Net \wedge (\exists c: Id \mid c = \{from, to\} \bullet Frame)$
$\wedge Switch\_Try \wedge Phones\_Ring[to/i] \wedge Phones\_GetRing[from/i]$

$Net\_Conn \triangleq$
$\Delta Net \wedge (\exists c: Id \mid c = \{from, to\} \bullet Frame)$
$\wedge Switch\_Connect \wedge Phones\_Answer[to/i] \wedge Phones\_GetConn[from/i]$

$Net\_Reject \triangleq \Delta Net \wedge (\exists c: Id \mid c = \{from\} \bullet Frame)$
$\wedge Switch\_Reject \wedge Phones\_GetBusy[from/i]$

$Net\_End \triangleq$
$\Delta Net \wedge (\exists c: Id \mid c = \{from, to\} \bullet Frame) \wedge Switch\_Terminate$
$\wedge (Phones\_Hangup[from/i] \wedge Phones\_LoseConn[to/i])$

$Net\_Initiate \triangleq \Delta Net \wedge (\exists c: Id \mid c = \{i\} \bullet Frame) \wedge Phones\_Initiate$

$Net\_Replace \triangleq \Delta Net \wedge (\exists c: Id \mid c = \{i\} \bullet Frame) \wedge Phones\_Replace$

Figure 4: Composition of Phones *view and* Switch *view*

A call ends when the caller hangs up, the called party loses the connection and the switch terminates the call:

$Net\_End \triangleq$
$\Delta Net \wedge (\exists c: Id \mid c = \{from, to\} \bullet Frame)$
$\wedge Switch\_Terminate \wedge (Phones\_Hangup[from/i]$
$\wedge Phones\_LoseConn[to/i])$

This is how British phones work; in the US, either party can terminate the call:

$Net\_End \triangleq$
$\Delta Net \wedge (\exists c: Id \mid c = \{from, to\} \bullet Frame)$
$\wedge Switch\_Terminate$
$\wedge ((Phones\_Hangup[from/i] \wedge Phones\_LoseConn[to/i])$
$\vee (Phones\_Hangup[to/i] \wedge Phones\_LoseConn[from/i]))$

5 More Complex Specifications

The editor and phone specifications exemplify two kinds of view composition. For the editor, the views were connected by an invariant relating their states; for the phone, the views were connected by synchronizing their operations, the states being related only indirectly.

Usually both forms of composition are needed. Suppose our phone switch offers screening list features. "Selective call rejection", for example, allows a subscriber to enter a list of phone numbers from which calls should always be rejected. In the *Phones* view , it is natural to represent screening lists as a function mapping valid subscriber identifier to a set of identifiers to be rejected:

$enemies: Id \pfun \mathbb{P}\, Id$

and to have a local operation to add a number such as:

Phones_Add

$\Delta Phones$
$i, e: Id$

$enemies'(i) = enemies(i) \cup \{e\}$
$ps(i) = dialtone \land ps'(i) = ps(i)$

In the *Switch* view, on the other hand, the screening list is more naturally represented as a relation between subscribers

$\_\ hates\ \_ : Id \leftrightarrow Id$

Now a call can be rejected even if the number being called is not busy, so we weaken the precondition of *Switch_Reject* to :

$(from, to) \in reqconns \land (to \in (\mathrm{dom}\ conns \cup \mathrm{ran}\ conns) \lor to\ hates\ from)$

and strengthen that of *Connect* by conjoining

$\ldots \land \neg (to\ hates\ from)$

Finally, the two representations of the screening lists are reconciled with an invariant:

PhoneSystem

Phones
Switch

$\forall i : \mathrm{dom}\ ps \bullet ps(i).enemies = hates\ (\!|\{i\}|\!)$

Conversely, an editor specification is likely to connect operations as well as states. Adding cursor memory (discussed in Section 2) required that the *csrLeft* and *csrRight* operations be represented in both *File* and *Grid* views. The event classification illustrated in the telephone example might be useful too. Scrolling, for example, might be activated by a mouse click, which is interpreted as either *pageUp* or *pageDown*, depending on where the mouse is clicked on a scroll bar.

This technique also eases the treatment of exceptions. An operation might be defined in several views even though the normal and erroneous cases are only distinguishable in one. Following standard Z practice, the operation would be split into two schemas, but these would not be combined within a view. Instead, the classification is propagated from one view (V) to the others (W_i) by conjunction, and only then are the normal and erroneous cases combined:

$$op \mathrel{\widehat=} (V\_opNormal \land W_1\_opNormal \land ...) \lor (V\_opError \land W_1\_opError \land ...)$$

6 Some Hints For View Structuring

To claim any kind of methodical procedure would be premature, but there are some symptoms of successful (and unsuccessful) view structuring that are worth pointing out.

A good view-structured specification should have simple and coherent views. Not only should each view's state space be simple; the operations themselves should be straightforward, without elaborate division into cases.

A strong (and maybe complicated) inter-view invariant is often a sign of an approriate separation; if the invariant is trivial, view structuring is unlikely to bring much benefit. Within a view, one should expect less complicated invariants than in a typical specification, and, in particular, a lack of projection functions, whose presence suggests an opportunity for more fine-grained view structuring.

A view should be partial, excluding unnecessary details of behaviour that will be resolved in other views, but it should not omit details for the sake of minimizing redundancy between views. View-structured specifications inevitably contain redundancies. In the phone specification, for example, the state of a telephone in the *Phones* view is enough to tell whether it is busy, but it would make no sense to rely on this and eliminate the precondition of the *Reject* operation in the *Switch* view.

A view should not export schemas that do not correspond to operations. One might be tempted, for example, to classify events in one view by conjoining to its operations preconditions specified in another view. This risks turning a view into an incoherent collection of predicates. The phone specification did export a framing schema, but this is hard to avoid in Z (see Section 8.2).

Ultimately, the most reliable indication of success is a clear partitioning of functionality: that, for any aspect of the function, it is clear to which view (if any) it belongs. How to achieve reliably a successful decomposition into views is the subject of future research; some tentative steps based on Michael Jackson's notion of "problem frames" [Jac94] are reported in [JJ95].

7 Why Z?

The choice of Z for our examples was not incidental; it has features that make it especially well suited to view structuring. The fundamental feature, of course, is schema composition. An operation can be formed from other operations by combining them with the standard logical connectives, and, likewise, a state space can be built from a collection of components and invariants. Schema composition is more than just allowing the use of logical connectives, however: it relies on a number of more subtle features in the language design.

7.1 Implicit Specification

An explicit specification specifies a state transition with assignments that construct a new value of the state in terms of its old value. An implicit specification merely states a relationship that must hold between new and old states. As a result, implicit specifications can rarely be executed. The compensating benefits usually cited are that specifications can be more concise and are less likely to embody premature implementation choices.

For our purposes, two different benefits are more vital. First, very partial specifications are possible, so we can specify a view that barely constrains a state component at all. (Often the cost of writing anything approaching a full specification is prohibitive anyway, and we want instead to model the intended behaviour of the system in only just enough detail to allow useful analysis. In that case, a single, very partial view may suffice.) Second, conjunction is available, so specifications can be composed: an operation can be specified to satisfy some constraints in one view and some different constraints in another. Implicit specification, then, is the fundamental basis of view structuring.

Even amongst the languages that encourage implicit specification, Z's pervasive use of invariants is unusual. A typical Z specification introduces the abstract state space with some strong invariants, and then gives far weaker specifications of operations than one might expect, leaving the invariants to work magic and fill in the details.

In Larch [GHW85], on the other hand, invariants play a secondary role, as a proof obligation rather than a specification mechanism. VDM [Jon90] likewise treats state invariants as proof obligations, although it allows type invariants to be specified (which are unnecessary in Larch because types are defined algebraically).

Since these languages allow conjunction in the pre- and post-conditions of operations, the Z style can always be imitated by conjoining invariants throughout. Usually, however, the operations are specified more constructively than in Z, with the specifier doing more work (and arguably, therefore, the reader doing less). Without inherited invariants and conjunction of operations, however, view structuring is not easily emulated.

7.2 Implicit Preconditions

The precondition of an operation is (roughly – more on this below) the set of states in which it may be invoked. In Z, preconditions are not specified directly. An operation is described by a single formula that characterizes a relation on states. The precondition characterizes the domain of this relation, and is a second formula that can be derived syntactically (but often not easily) from the first.

For a simple operation that is not composed from other operations, it makes little difference whether the precondition is implicit as in Z, or explicit as in Larch and VDM. In a proof of refinement, contravariance – sanction to broaden the precondition but narrow the postcondition – makes it necessary anyway to make the precondition explicit. (If the operation is specified as $pre \Rightarrow post$, in Hoare's style [Hoa85], rather than $pre \wedge post$, in Z style, the natural contravariance of implication allows the precondition to remain implicit, and the refinement check reduces to $impl \Rightarrow spec$.)

In view structuring, it is essential that the precondition be implicit. When two operations are conjoined, the new operation will have a precondition that is at least as strong as the conjunction of their preconditions, but in some cases stronger. There may be pre-states for which the two operations cannot agree on any appropriate post-states; these must then be ruled out. Put formally, the precondition calculation does not distribute over conjunction, so the implication in

$$pre\ (op_1 \wedge op_2) \Rightarrow pre\ (op_1) \ \wedge \ pre\ (op_2)$$

does not apply in the other direction. This means that if pre- and postconditions were specified separately, they could not be conjoined pairwise, but rather the precondition of the combined operation would be a complicated expression involving the pre- and post-conditions of both operations [War93].

7.3 Explicit Frame Conditions

In most specification languages, any variable not mentioned in an operation is assumed be unchanged by its execution. Consider the modification clause of Larch, for instance. The specification of a procedure with the arguments x, y and z might say "modifies at most x", indicating that y and z, and certainly any global variables, are invariant. A Z schema, in contrast, would include $y' = y \wedge z' = z$. Any state variable that is not explicitly constrained is free to change arbitrarily.

For view structuring, implicit frame conditions that say "and nothing else changes" are catastrophic. When specifying a view we cannot know which other views will later be written, or what their variables will be. Inter-view invariants ensure that modification of the variables of one view almost always propagates to those of another.

Views could benefit from a different kind of frame condition. Sometimes a view has local variables that are not related to the variables of another view, and are thus unconstrained by the inter-view invariant. A phone might have a volume control, for example, that is unaffected by the operations of another phone or of the switch. To say this, the volume control must awkwardly be brought into the scope of those operations. A better solution employs a frame condition, declaring the volume to be a

local component of the *Phone* view that cannot change except by execution of local operations. This is a special case of a more general scheme developed by Borgida et al [BMR93].

7.4 Uniform Data Representation

Z really has only two datatypes – sets and tuples – so with tuples expressed by schema signatures, a powerset operator alone would suffice to generate all types. The wide variety of types (total, partial, injective and surjective functions; relations, sequences, etc.) is a syntactic convenience, since each can be reduced to sets and tuples with an appropriate constraint.

This makes it easy to express inter-view invariants, since no type coercions are required. Suppose, for example, a sequence in one view is to be constrained to have the same elements as a set in another view. In a language that defines sequences and sets as different types, each with its own algebraic theory, it would be necessary to define a function that constructs a set from a sequence. In Z, however, a sequence of elements of type T is a function from a prefix of the naturals to T, and the set of elements appearing in a sequence s is just $ran\ s$.

8 Why Not Z?

For the purpose of view structuring, Z has a number of deficiencies. One of these – the interpretation of preconditions – is intolerable, but luckily, no linguistic (or arguably even semantic) issues are at stake, and the problem can be overcome by a shift in interpretation. The second – the lack of indexing – is inconvenient but can be worked around. The third – the lack of true actions – makes it harder to express view structuring naturally, and rules out more ambitious ways to relate views.

8.1 Preconditions as Disclaimers

In all the specification languages that grew out of work in abstract data types, the precondition of an operation is a *disclaimer*. Should the operation be invoked in a state not satisfying the precondition, the specification makes no promises about the subsequent behaviour.

Some approaches view the disclaimer as part of the operation's behavioural specification: the operation's semantics will then generally allow it, when invoked in a bad state, only to return in an arbitrary state or never return at all. In other approaches, the disclaimer is a proof obligation attached to the operation but imposed on callers. This more accurately models the use of disclaimers in code, admitting disasters beyond the formal model, such as corruption of the state of another process. (Maibaum's deontic logic brings these approaches together, making explicit and formal not only the effect of executing an operation but also the obligations of the system as a whole to execute or not execute it [Mai93].)

Languages that grew out of Dijkstra's guarded commands have a different notion of precondition. In CSP, for example, the process

$$a \rightarrow c \rightarrow P \mid b \rightarrow d \rightarrow Q$$

allows only the event c following the event a. One cannot ask what happens if d is invoked following a; it simply cannot happen. This kind of precondition is called a *guard* or *firing condition*.

Z is an odd hybrid of the these two approaches. In the semantics, an operation schema denotes a relation on states. One can then think of the entire specification as a state machine in which these transition relations are overlaid, each being a set of arcs between states labelled with the name of the operation. The precondition of an operation is the set of states with outgoing arcs labelled with the operation's name. One might thus assume that Z follows the CSP approach, and that an operation cannot occur in a state for which the precondition is false.

The standard interpretation of Z is determined, however, by its refinement rules. These allow an implementation to widen the precondition, in line with the view of precondition as disclaimer. (The CSP trace semantics, in contrast, effectively allows the precondition to be narrowed.) Unfortunately, for the style of specification advocated here, it is essential that preconditions be viewed as guards and not disclaimers. There are two reasons.

First is our use of preconditions to classify events. In the telephone specification, for example, whether lifting the phone constitutes an *Answer* event or an *Initiate* event is determined by the state of the telephone (whether it is ringing or not). It makes no sense to ask what happens if the user "invokes" an *Answer* event when the phone is not ringing.

Second, we have specified events that are not invoked by a user but are performed autonomously by the system. Weakening the precondition of the *Reject* operation of our phone switch (Fig. 3), as sanctioned by Z refinement rules, would allow the switch to reject a requested connection even when the number being called is not busy!

Interpreting preconditions as guards is no great loss. An implementor of an operation will still be able to treat the precondition as a disclaimer, knowing that whatever schedules the operation will prevent its execution in bad states. And by weakening the specification to

$$guard \wedge (disclaimer \Rightarrow post)$$

the specifier can still leave the state transition unconstrained even in cases in which the operation can occur.

Although the refinement rules of Z do not apply, the semantics of the specification is arguably unaltered. The notion that "anything can happen" when an operation is invoked in a bad state does not appear in the semantics, and is more cultural than technical, there being no way to express non-terminating behaviour or other disasters in Z anyway.

8.2 Lack of Indexing

In our telephone specification, it would have been more natural to model each phone as a view it its own right. Unfortunately, this is not easily accomplished. Having defined the state of a phone view as

Phone

s: *Status*

say, it is straightforward to include an indexed set of phone states in the global state

Net

ps: $Id \pfun Phone$

Switch

but combining operations proves far more tricky. The operations of the phone view operate on a single phone, and to apply them to the new global state, they must first be "promoted". The definition of *Net_DialReq*, for example, would become

$$\begin{aligned} &Net\_DialReq \mathrel{\widehat=} \\ &\quad \Delta Net \land (\exists c: Id \mid c = \{from\} \bullet Frame) \\ &\quad \land Switch\_Request \land \exists \Delta Phone \bullet Bind \land Phone\_Dial[from/i] \end{aligned}$$

where the promotion schema

Bind

ΔNet

$\Delta Phone$

i: Id

$ps(i) = s \land ps'(i) = s'$

is essentially a wrapper that converts a state change on *Phone* to a change on *Net*.

This is cumbersome and a bit obscure. It would be far more natural to define the global state as a collection of *Phone* views and a *Switch* view

Net

$Phone(i: Id)$

Switch

and expose the arguments of the operations, writing something like

$$\begin{aligned} &Net\_DialReq(from, to) \mathrel{\widehat=} \\ &\quad \Delta Net \land Switch\_Request\ (from, to) \\ &\qquad \land Phone(from)\_Dial\ (to) \land \forall i \neq from \bullet \Xi Phone(i) \end{aligned}$$

where *Phone(from)_Dial(to)* denotes the *Dial* operation of the *Phone* view with index *from*, with its argument bound to the identifier *to*. This is not legal Z, of course. Indexing is not allowed, and for good reason: it breaks the simple syntactic nature of Z.

Our example illustrates, however, that its omission comes at a serious cost. The specifier has to construct the indexing explicitly with functions, often resulting in a mass of confusing hidings and renamings. The resulting temptation to expand a Z schema by writing it out in full is a sure sign of a breakdown of modularity, in which schemas have become little more than macros.

8.3 Lack of Actions

An operation in Z is equated with its transitions; the name of the schema has no semantic significance. The definition

$$Net\_DialReq \cong \ldots Switch\_Request \wedge Phones\_Dial[from/i] \ldots$$

thus says nothing more than that the transitions associated with the name *Net_DialReq* are obtained by conjoining the formulae named by the symbols *Switch_Request*, *Phones_Dial*, and so on. Our intuitive reading has no basis in the formalism: the definition does not say that a *Net_DialReq* event is the simultaneous occurrence of the events *Switch_Request* and *Phones_Dial*.

This is more than a philosophical problem, because it means we must constantly be on guard, checking that, in every assertion we write down, our intuition is supported by the formal interpretation. Often it will not be. Suppose we want to demonstrate that whenever the *Net_DialReq* event occurs, some phone performed a *Phones_Dial* event. We might be tempted to write something like

$$Net\_DialReq \Rightarrow \exists i: Id \bullet Phones\_Dial$$

but unfortunately, this says nothing of the sort: it claims only that every state transition matching the *Net_DialReq* schema also matches the *Phones_Dial* schema, and might be true even if *Net_DialReq* were constructed from other events in the *Phones* view.

8.4 Closed vs. Open Specification

A Z specification is "closed": once schemas have been written, nothing more can be said about them. The only way to extend a specification is to add new schemas that incorporate the old ones.

View structuring seems to call for a more "open" specification model. The composition of two views might then be thought of not as the creation of a new specification object, but rather the addition of new properties relating existing views. To say that dialling at a phone is associated with the registering of a request at the switch, we could then add an assertion of the form

$$Phones\_Dial \Leftrightarrow Switch\_Request$$

without a need to invent a new (and spurious) action such as *Net_DialReq*.

Combining this idea with real actions and indexing might result in a formalism more directly suited to view structuring. Suppose we associate with each transition zero or more action names, and write

$[a]$

for the elementary logical assertion "the transition is labelled with the action name a". Then the assertion

$[invest(i)] \Rightarrow bal' = bal + i$

says that when the *invest* action occurs with argument i, the balance is increased by i, while

$bal' > bal \Rightarrow [credit]$

says that any transition in which the balance increases is to be classified as a *credit* event. From these we can deduce

$[invest(i)] \wedge (i > 0) \Rightarrow [credit]$

namely that all *invest* actions of positive amounts are *credit* events too.

Returning to the dialling example, we can dispense with the composite action *Net_DialReq* and write

$[Phone(from)\_Dial(to)] \Leftrightarrow [Switch\_Request(from, to)]$

which says exactly what we want: that each occurrence of a dial action at a phone corresponds to a request action at the switch, and vice versa. To constrain no action to occur at any other phone at the same instant, we would add

$[Phone(from)\_Dial(to)] \Rightarrow \forall i: Id \mid i \neq from \bullet \Xi Phone(i).$

9 Related Work

9.1 Views in Conventional Z Specifications

View structuring can be seen, to a greater or lesser extent, in many published Z specifications. When the structure of the state variables makes the description of an operation awkward, the Z specifier will frequently add redundant variables – tied by an invariant to the existing variables – and constrain them instead. For example, a specification of a program for allocating resources to users at various times includes in its state not only the full relation between resources, users and times, but also various projections, such as a relation between resources and times, appropriate for describing operations, like checking availability, that are not concerned with the users [FS93]. The same specification employs the conjunction of operations on different state spaces.

Sufrin's specification of a text editor [Suf82] comes closest to full view-structuring. He sidesteps the problem of defining the effect of editing operations on the text's screen appearance by specifying all the operations over a simple representation, which is then related to the layout on the screen by an invariant. But these represen-

tations are not views. In a view-structured specification, neither representation is primary, and the two views stand alone as specifications in their own right. In Sufrin's specification, the display representation has no associated operations, so the editor provides only left and right cursor movements, for example, but not up or down. Furthermore, displaying is regarded as an operation called *show* whose specification is the invariant that would have related the two views. Nevertheless, Sufrin has views in mind; in discussing related work he mentions the possibility of multiple representations related implicitly by invariants, each with its own operations.

Promotion [MS84], a common Z structuring technique, may also be seen as a limited kind of view structuring. A library specification might define the state of a book with operations such as *lend* and *return*. The entire library might then be modelled as a mapping from identifiers to books. To specify the system operation corresponding to a *lend* of a book with a particular identifier, the book operation is "promoted" to the state of the entire system. The book operations do not constitute a view, however, since they are regarded as part of the larger system.

Most Z specifications are not view-structured in any sense. Whole and part composition dominates. Frequently, not only the structure of the specification, but also the structure of the development itself, is hierarchical. Woodcock talks about the "onion skins of software development", in which a specification is developed from the inside out, with promotion at the interfaces between the layers [Woo89].

Not all systems are easily described in this fashion. While a file system may contain files, a telephone switch does not "contain" telephones in any sense. So telephone operations cannot be simply promoted. Without views, however, the behaviour of a single telephone cannot be separated from the behaviour of the switch, and the two become intertwined. Woodcock's telephone specification [WL88] embeds the states of the individual phones in the global state of the telephone system, so that every action of a subscriber becomes an action of the whole system, and the observable behaviour at a given phone is relegated to theorems.

9.2 Views in Other Specification Languages

Property-oriented specifications often support structuring mechanisms akin to views. In the Larch Shared Language [GHM90], the basic unit of specification is a "trait". Several traits can assert different properties of the same operators, and then be combined into a single trait. A queue, for example, may be specified in two traits: one that asserts basic container properties, and another expressing the FIFO ordering. A specification of a set can share the container trait, combining it instead with a trait expressing the notion of single occurrence of each element. Trait composition is not used to structure operations, however; one of the main contributions of Larch has been to isolate the basic properties of types (defined algebraically) from execution concerns like preconditions, termination and exceptions (defined in predicate calculus in a separate tier). The problems of representing partiality and non-determinism algebraically are thus side-stepped.

9.3 Viewpoints: Consistency and Amalgamation

The separation of a specification into multiple views has been advocated many times before, for various applications. Currently, three British groups – at Imperial College,the University of Bath, and the University of Kent – are investigating the notion of "viewpoints".

Their approach differs from mine in primarily two respects. First, they focus more on the activities of checking consistency between viewpoints and amalgamating them; this paper has been concerned instead with the structuring of the specification. Second, they assume a structural correspondence between viewpoints, so that the relationship between them need not be specified but rather may be inferred by the use of common names. The view structuring proposed here, in contrast, exploits complex relationships between views to simplify the views themselves.

The viewpoints of the Imperial College group [F&92] arise from the different domains of developers working together on a team. The developers of a lift system, for instance, will include user-interface designers, mechanical engineers and real-time experts; some will be concerned with performance, some with functionality, some with safety, and so on. Clearly these viewpoints are so disparate that to attempt a coherent model would be absurd: far better that each developer work within a viewpoint, with its own notations, models and tools.

Inconsistency between viewpoints during a software development is regarded, in this approach, much like inconsistency of database records during a transaction. Although it must ultimately be resolved, they will inevitably be periods during which it should be tolerated. Instead of enforcing consistency, therefore, the relationship between viewpoints is monitored, and detected inconsistencies are reported to users. There is no universal model into which viewpoints are translated; instead, global syntactic checks between viewpoints are triggered by local changes [EN95]. Viewpoint construction is thus a kind of cooperative work (with problems similar to the joint editing of any document) whose distributed nature is essential.

The viewpoints of the Kent group [DBS95] also arise from the involvement of participants with differing perspectives. The viewpoints of the Bath group [A&94], in contrast, are proposed as a means of structuring large specifications. Despite their different motivations (the latter being closer to mine), both groups have investigated the use of Z for specifying viewpoints and see "amalgamation" (or "unification") as the primary issue. They argue that the division into viewpoints must eventually be abandoned, to suit programmers, for example, who want a single specification of an operation.

Each view is treated as a partial specification. The problem of amalgamation is to construct a complete specification that is a refinement of the individual views. Any implementation of the complete specification will then satisfy each view specification independently. Unfortunately, because of the contravariance inherent in operation specifications, conjunction does not preserve refinement in Z; a refinement of

$$op_1 \wedge op_2$$

is not generally a refinement of both op_1 and op_2. The complete specification cannot

be obtained just by conjoining the operations of the views. The Bath group loosen the notion of refinement, and suggest that different combination mechanisms may be suitable in different cases. The Kent group stick to the conventional notion of refinement in Z, and show to combine operations to give the weakest complete specification that is a refinement of the individual views.

A fundamental assumption of both groups is that there is a simple structural correspondence between the views. Operations (and similarly state components) in different views that share the same name are matched. My view structuring makes no such assumptions, and relies on the specifier to make explicit the relationship between views. It seems that the greatest benefit of view structuring comes when this relationship is complicated. The inter-view invariant of the editor (Fig. 1) allowed radically different representations for the two views; the complex composition of operations in the phone switch (Fig. 4) allowed us to associate the ringing of one phone, for example, with a ringtone in another.

Relationships between views can sometimes be eliminated by adding extra views that incorporate them. Complex composition of operations cannot be handled in this way, however, without using the operations of one view inside another, which violates the separation of views.

9.4 Descriptions

Zave and Jackson propose a structuring mechanism for specifications whose units they call "descriptions" [ZJ93]. They show how a number of specification notations can be translated into a minimal language based on first-order logic, in an assertional style similar to that proposed in Section 7.3. Descriptions are then combined simply by conjunction.

The value of multiple paradigms is indisputable; the phone view of Figure 2, for example, would have been far more naturally expressed in a finite transition diagram. But there is price to pay for using more than one notation. Reasoning about the composition, which is already difficult in view structuring, becomes even harder. Moreover, the various notations must sometimes be given unconventional interpretations to allow them to be integrated smoothly. On the other hand, by allowing many notations it becomes possible to reduce each to its bare essentials, with a simpler and more transparent semantics than would suffice for a single paradigm approach.

Interpreting an action (such as lifting the handset) as one of several events (answering or initiating a call) depending on context (whether the phone is ringing) is fundamental to view structuring, especially when the context of one view disambiguates the events of another. This idea originates in Zave's work on telephone specifications [Zav85] and has been developed into the notion of "event classification" in her colaboration with Jackson [ZJ94]. Full event classification requires true actions (see Section 8.3), and can only be weakly imitated in Z by forming composite operations (as in Figure 4).

9.5 Views in Programming and Environments

Nord treats a similar problem at the programming level [Nor92]. He argues (using the ubiquitous editor example) that implementing a type can be much easier if multiple representations are allowed. His scheme has the programmer code each operation in the most convenient representation, and assert invariants between the diverse representations. A single representation is obtained from these multiple representations by first forming their cross-product, and then applying semi-automatic program transformation techniques to derive code to maintain the relationship between components incrementally.

Nord's focus on programs rather than specifications means that his composition mechanism is more limited, so he cannot allow, for instance, an operation that appears in two views. Nevertheless, these ideas raise an interesting prospect of implementing view-structured specifications without amalgamating them first.

Multiple views have been also proposed for the state shared by the tools of a programming environment. In Garlan's scheme [Gar87] for example, the relationships between representations are inferred from their type structure, and update algorithms are synthesized automatically. A set in one view and a sequence in another, for example, would be assumed to contain the same elements if they had the same name. (In this work's extension in the Janus project [H&88], new mappings could be specified by the environment's developer, but were still selected on the basis of type.)

10 Summary

Structuring a specification in views has many benefits. Separating different aspects of the function of a system into different views allows each to be expressed in its most natural representation. Since a view is a partial specification of the entire system, it can be evaluated directly against the perceived requirements, and can be constructed and analyzed independently of other views. The complexities of interaction between different functions may be deferred until a later stage, when the views are connected. A view-structured specification is easier to maintain because a change to only one aspect of functionality can often be confined within a view.

Z is especially well-suited to view structuring. The vital features are: implicit specification, implicit preconditions, explicit frame conditions and uniform data representation. Other features of Z – the lack of true actions and indexing, and the interpretation of preconditions as disclaimers – frustrate view structuring but can often be overcome, albeit inelegantly.

Despite its benefits, view-structuring has yet to be fully exploited. Although evident in many Z specifications, it has not been systematically applied. The examples in this paper have attempted to demonstrate that views offer a separation of concerns that is widely applicable, and that view structuring could profitably be added to the specifier's repertoire.

Acknowledgments

Gregory Abowd, Michael Jackson, Jeannette Wing, Michal Young and Jim Woodcock gave me helpful comments on early drafts of this paper; many thanks to them all. I am especially grateful to Pamela Zave and the anonymous referees for their detailed criticism of the submitted version.

References

[A&94] M. Ainsworth, A.H. Cruickshank, L.J. Groves and P.J.L. Wallis, "Viewpoint Specification and Z", *Information and Software Technology*, 1994, 36(1), pp. 43–51.

[BMR93] Alex Borgida, John Mylopoulos and Raymond Reiter, "And Nothing Else Changes: The Frame Problem in Procedure Specifications", *Proc. 15th International Conference on Software Engineering*, Baltimore, Maryland, IEEE Computer Society Press, May 1993.

[DBS95] J. Derrick, H. Bowman and M. Steen, "Maintaining Cross-Viewpoint Consistency using Z", *IFIP International Conference on Open Distributed Processing*, Chapman and Hall, 1995, pp. 395–406.

[EN95] S. Easterbrook and B. Nuseibeh, "Managing Inconsistencies in an Evolving Specification", Proc. Second IEEE International Symposium on Requirements Engineering, York, England, March 1995.

[F&92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development", *International Journal on Software Engineering and Knowledge Engineering*, 2(1):31–57, World Scientific Publishing Company, March 1992.

[FS93] Bill Flinn and Ib Holm Sorensen, "Caviar: A Case Study in Specification", Chap. 5 of [Hay93].

[Gar87] David Garlan, *Views for Tools in Integrated Environments*, Technical Report CMU-CS-87-147, School of Computer Science, Carnegie Mellon University, May 1987.

[GHM90] John V. Guttag, James J. Horning and Andres Modet, *Report on the Larch Shared Language: Version 2.3*, Technical Report 58, Digital Systems Research Center, Palo Alto, CA, April 1990.

[GHW85] John Guttag, James Horning and Jeannette Wing, "The Larch Family of Specification Languages", *IEEE Software*, Sept. 1985.

[H&88] A.N. Habermann, Charles Krueger, Benjamin Pierce, Barbara Staudt and

John Wenn, *Programming with Views*, Technical Report CMU-CS-87-177, School of Computer Science, Carnegie Mellon University, January 1988.

[Hal93] Anthony Hall, "A Response to Florence, Dougal and Zebedee", *FACS Europe (Newsletter of the British Computing Society Formal Aspects of Computing Science Special Interest Group and Formal Methods Europe), Series 1, Vol. 1, Num. 1, Autumn 1993.*

[Hay92] Ian Hayes, "VDM and Z: A Comparative Case Study", *Formal Aspects of Computing*, Vol. 4, No. 1, 1992, Springer International.

[Hay93] Ian Hayes, ed., *Specification Case Studies*, Prentice Hall, second edition, 1993.

[HJN93] I.J. Hayes, C.B. Jones and J.E. Nicholls, "Understanding the Differences Between VDM and Z", *FACS Europe (Newsletter of the British Computing Society Formal Aspects of Computing Science Special Interest Group and Formal Methods Europe), Series 1, Vol. 1, Num. 1, Autumn 1993.*

[Hoa85] C.A.R. Hoare, "Programs are predicates", in *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson (eds.), pp. 141–154, Prentice-Hall International, 1985.

[Jac94] Michael Jackson, "Software Development Method", in *A Classical Mind: Essays in Honour of C.A.R Hoare*, ed. A.W. Roscoe, Prentice Hall International, 1994.

[JJ95] Daniel Jackson and Michael Jackson, "Problem Decomposition for Reuse", Technical Report CMU-CS-TR-95-108, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1995.

[Jon90] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International, second edition, 1990.

[Mai93] Tom Maibaum, "Temporal Reasoning over Deontic Specifications", in *Deontic Logic in Computer Science: Normative System Specification*, ed. John-Jules Ch. Meyer and Roel J. Wieringa, Wiley, England, 1993.

[Mor93] Carroll Morgan, "Telephone Network", Chap. 3 of [Hay93].

[MS84] C.C. Morgan and B.A. Sufrin, "Specification of the UNIX Filing System", *IEEE Transactions on Software Engineering*, SE-10(2), 1984.

[Nor92] Robert L. Nord, *Deriving and Manipulating Module Interfaces*, Technical Report CMU-CS-92-126, School of Computer Science, Carnegie Mellon University, May 1992.

[Spi92] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, second edition, 1992.

[Suf82] Bernard Sufrin, "Formal Specification of a Display-Oriented Text Editor", *Science of Computer Programming*, 1, pp 157–202.

[War93] Nigel Ward, "Adding Specification Constructors to the Refinement Calculus", *Proc. of FME '93: First International Symposium of Formal Methods Europe*, Odense, Denmark, April 1993. Lecture Notes in Computer Science, Number 670, Springer-Verlag , pp. 652–670.

[Woo89] J.C.P Woodcock, "Mathematics as a Management Tool: Proof Rules for Promotion", *Proc. of the Centre for Software Reliability Conference entitled "Large Software Systems"*, Bristol, UK, Sept. 1989.

[WL88] Jim Woodcock and Martin Loomes, "Case Study: A Telephone Exchange", Chap. 9 of *Software Engineering Mathematics*, Addison-Wesley, 1988.

[Zav85] Pamela Zave, " A Distributed Alternative to Finite-State-Machine Specification", ACM Transactions on Programming Languages and Systems 7(1):10–36, January 1985.

[ZJ93] Pamela Zave and Michael Jackson, "Conjunction as Composition", *ACM Trans. on Software Engineering and Methodology*, 2(4), pp. 379–411, Oct. 1993.

[ZJ94] Pamela Zave and Michael Jackson, *Where Do Operations Come From? A Multiparadigm Specification Technique*, Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ, April 94.