Computer Science

Structuring Z Specifications with Views

Daniel Jackson
March 1994
CMU-CS-94-126

Carnegie Mellon

Structuring Z Specifications with Views

Daniel Jackson
March 1994
CMU-CS-94-126

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A view is a partial specification of a program, onsisting of a state space and a set of operations. A full specification is obtained by composing several views, linking them though 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. The features of Z that make it especially well suited to composing views are discussed, along with some hints for adapting other languages to the purpose.

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 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 VDM by checking implementability).

Why not extend the benefits of implicit specification beyond the definition of operations? This paper proposes a structuring mechanism at the grossest level, where separation of concerns has the same virtues but a different flavour. The program is specified as the conjunction 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.

The seeds of view structuring are evident in many Z published 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). Other languages, such as VDM, tend to eschew the kind of redundancy and pervasive use of invariants favored by Z, so these techniques are rarely seen elsewhere.

This paper contains no radical novelties. Its intent is to articulate, by means of small illustrations, 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].

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

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

The operation that moves the cursor forward in the text can now be written as:

csrRight

$\Delta\ File$

$right \neq \langle\rangle$
$left' = left \frown \langle head(right) \rangle$
$right' = tail(right)$

and the insertion of a character as:

$$\begin{array}{|l}
\textit{insertChar} \\
\hline
\Delta \textit{ File} \\
c?: \textit{Char} \\
\hline
\textit{left}' = \textit{left} \frown \langle c? \rangle \wedge \textit{right}' = \textit{right} \\
\hline
\end{array}$$

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 soft line break in *left*, which in turn depends on the placement of word separators in the entirety of *left*!

A better state representation for the *csrUp* operation is a sequence of lines, with a cursor given as a coordinate pair:

$$\begin{array}{|l}
\textit{Grid} \\
\hline
\textit{lines}: \textit{seq seq Char} \\
x, y: \mathbb{N} \\
\hline
y \in \mathrm{dom}\ \textit{lines} \ \wedge\ x \in \mathrm{dom}\ \textit{lines}[y] \\
\hline
\end{array}$$

Moving the cursor up is now easy to define

$$\begin{array}{|l}
\textit{csrUp} \\
\hline
\Delta \textit{ Grid} \\
\hline
y > 1 \wedge y' = y - 1 \\
x' = \textit{min}\ (x, \#\textit{lines}[y']) \\
\textit{lines}' = \textit{lines} \\
\hline
\end{array}$$

as are other line-based operations, such as deleting from the cursor to the end of the line:

$$\begin{array}{|l}
\textit{delEol} \\
\hline
\Delta \textit{ Grid} \\
\hline
\textit{lines}'[y] = \textit{lines}[y][1..x] \\
\{y\} \lhd\!\!\!- \textit{lines}' = \{y\} \lhd\!\!\!- \textit{lines} \\
x' = x \ \wedge\ y' = y \\
\hline
\end{array}$$

Automatic text wrapping is easily defined on this representation too. A line is legal if it ends with a newline or a space, and has no internal carriage returns:

$$\begin{array}{|l}
\textit{legalLines}: \mathbb{P}\ \mathrm{seq}\ \textit{Char} \\
\hline
\forall l: \mathrm{seq}\ \textit{Char} \bullet \\
\quad l \in \textit{legalLines} \Leftrightarrow \forall i \in \mathrm{dom}\ l.\ (l[i] = \backslash n \Rightarrow i = \#l) \ \wedge\ l[\#l] \in \{\backslash n, \textit{spc}\}
\end{array}$$

and a sequence of lines is wrapped if every constituent line is legal and no longer than some maximum length:

$$
\begin{array}{|l}
\mathit{wrapped}: \mathbb{P}\ \mathrm{seq}\ \mathrm{seq}\ \mathit{Char} \\
\hline
\forall \mathit{ls}: \mathrm{seq}\ \mathrm{seq}\ \mathit{Char} \bullet \\
\quad \mathit{ls} \in \mathit{wrapped} \Leftrightarrow \mathrm{ran}\ \mathit{ls} \subseteq \mathit{legalLines} \wedge \forall l \in \mathrm{ran}\ \mathit{ls} \bullet \#l \leq \mathit{maxlinelen}
\end{array}
$$

This representation, however, is no good for the previous operations. Inserting a character is a complicated affair; despite the text wrapping invariant, we still have to place the inserted character, and are dragged into clumsy reasoning about what happens near the end of a line.

The solution to this dilemma is to have our cake and eat it. The specification is divided 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. First, we state that the concatenation of all the lines in the *Grid* view is the concatenation of the left and right portions of the text in the *File* view, and that the length of the left portion in the *File* view is the sum of the lengths of the line segments in the *Grid* view up to the cursor:

$$
\begin{array}{|l}
\hline
\mathit{Flatten} \\
\mathit{File}, \mathit{Grid} \\
\hline
\mathit{left} \frown \mathit{right} = {\frown}\mathit{lines} \\
\#\mathit{left} = x + \Sigma\, i{:}1..y-1 \bullet \#\mathit{lines}[i] \\
\hline
\end{array}
$$

Additionally, the division into lines should be optimal, so that if a word can fit on a line, it will not be wrapped. One division is better than another if its first line is longer, or if its first line is the same length and the division of the remaining lines is better:

$$
\begin{array}{|l}
>: \mathrm{seq}\ \mathrm{seq}\ \mathit{Char} \leftrightarrow \mathrm{seq}\ \mathrm{seq}\ \mathit{Char} \\
\hline
\forall \mathit{ls}_1, \mathit{ls}_2: \mathrm{seq}\ \mathrm{seq}\ \mathit{Char} \bullet \\
\quad \mathit{ls}_1 > \mathit{ls}_2 = (\#\mathit{head}(\mathit{ls}_1) > \#\mathit{head}(\mathit{ls}_2)) \\
\qquad \vee (\#\mathit{head}(\mathit{ls}_1) = \#\mathit{head}(\mathit{ls}_2) \wedge \mathit{tail}(\mathit{ls}_1) > \mathit{tail}(\mathit{ls}_2))
\end{array}
$$

Now we can state the invariant fully. Any division into lines that also matches the representation as a flat sequence must be no better than the actual division:

$$
\begin{array}{|l}
\hline
\mathit{Editor} \\
\mathit{Flatten} \\
\hline
\forall \mathit{ls}: \mathrm{seq}\ \mathrm{seq}\ \mathit{Char} \bullet \mathit{Flatten}[\mathit{ls}/\mathit{lines}] \Rightarrow \neg\, (\mathit{ls} > \mathit{lines}) \\
\hline
\end{array}
$$

File

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

File.csrRight

$\Delta File$

$right \neq \langle\rangle \ \wedge\ right' = tail(right)$
$left' = left \frown \langle head(right) \rangle$

File.insertChar

$\Delta File$
$c?: Char$

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

Grid

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

$lines \in wrapped \ \wedge\ y \in \mathrm{dom}\ lines \ \wedge\ x \in \mathrm{dom}\ lines[y]$

Grid.csrUp

$\Delta Grid$

$y > 1 \ \wedge\ y' = y - 1 \ \wedge\ x' = min(x, \#lines[y'])$
$lines' = lines$

Grid.delEol

$\Delta Grid$

$lines'[y] = lines[y][1..x]$
$\{y\} \ndres lines' = \{y\} \ndres lines$
$x' = x \ \wedge\ y' = y$

Flatten

$File, Grid$

$left \frown right = \frown/ lines \ \wedge\ \#left = x + \Sigma i{:}1..y-1 \bullet \#lines[i]$

Editor

$Flatten$

$\forall ls: \mathrm{seq}\ \mathrm{seq}\ Char \bullet Flatten[ls/lines] \Rightarrow \neg\ (ls > lines)$

$insertChar = [\Delta Editor \mid File.insertChar]$
$csrRight = [\Delta Editor \mid File.csrRight]$
$csrUp = [\Delta Editor \mid Grid.csrUp]$
$delEol = [\Delta Editor \mid Grid.delEol]$

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

Finally, the operations of the whole program are defined, each being taken from the appropriate view:

$insertChar = [\Delta Editor \mid File.insertChar]$
$csrRight = [\Delta Editor \mid File.csrRight]$
$csrUp = [\Delta Editor \mid Grid.csrUp]$
$delEol = [\Delta Editor \mid Grid.delEol]$

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.

The specification fragments are brought together in Fig. 1. The benefit of two representations is clear – a dubious reader might try to recast an operation from one view in the representation of another. To see how views give a helpful modularity, consider a few likely modifications.

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 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, having 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 this 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.

More elaborate features might require new views. A *Word* view for specifying spelling checks might represent the buffer as a sequence of words, abstracting away 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.

$[Id]$

$Onhook = \{ringing, idle\}$
$Tones = \{dialtone, ringtone, busytone\}$
$Status = Onhook \cup Tones \cup \{waiting, connected\}$

$Phone = [status: Status]$

$Phone.Initiate = [\Delta Phone \mid status = idle \land status' = dialtone]$
$Phone.Dial = [\Delta Phone; to?: Id \mid status = dialtone \land status' = waiting]$
$Phone.GetRing = [\Delta Phone \mid status = waiting \land status' = ringtone]$
$Phone.GetBusy = [\Delta Phone \mid status = waiting \land status' = busytone]$
$Phone.GetConn = [\Delta Phone \mid status = ringtone \land status' = connected]$
$Phone.LoseConn = [\Delta Phone \mid status = connected \land status' = dialtone]$
$Phone.Hangup = [\Delta Phone \mid status = connected \land status' = idle]$
$Phone.Replace = [\Delta Phone \mid status = dialtone \land status' = idle]$
$Phone.Answer = [\Delta Phone \mid status = ringing \land status' = connected]$
$Phone.Ring = [\Delta Phone \mid status = idle \land status' = ringing]$

Figure 2: The Phone *view*

3 Joining Views by their Operations

In the editor specification (Fig. 1), 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 with states like *idle*, *dialtone*, *waiting*, etc. Each operation corresponds to a transition, so lifting the handset when the phone is idle, for example, can be modelled by

$$\begin{array}{|l}
\multicolumn{1}{l}{Initiate} \\
\Delta Phone \\
\hline
status = idle \land status' = dialtone \\
\hline
\end{array}$$

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

$$\begin{array}{|l}
\multicolumn{1}{l}{Answer} \\
\Delta Phone \\
\hline
status = ringing \land status' = connected \\
\hline
\end{array}$$

The separation of the same action (lifting the handset) into two different classes of operation is determined locally by the status of the phone when the

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Switch}} \\
reqconns: Id \leftrightarrow Id \\
conns: Id \rightarrowtail Id \\
\hline
conns \subseteq reqconns
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Switch.Request}} \\
\Delta Switch \\
from?, to?: Id \\
\hline
reqconns' = reqconns \cup \{(from?, to?)\} \\
conns' = conns
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Switch.Connect}} \\
\Delta Switch \\
from?, to?: Id \\
\hline
(from?, to?) \in reqconns \;\wedge\; to? \notin (\mathrm{dom}\ conns \cup \mathrm{ran}\ conns) \\
conns' = conns \cup \{(from?, to?)\} \\
reqconns' = reqconns
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Switch.Reject}} \\
\Delta Switch \\
from?, to?: Id \\
\hline
(from?, to?) \in reqconns \;\wedge\; to \in (\mathrm{dom}\ conns \cup \mathrm{ran}\ conns) \\
reqconns' = reqconns \setminus \{(from?,to?)\}
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Switch.Terminate}} \\
\Delta Switch \\
from?, to?: Id \\
\hline
(from?, to?) \in conns \\
conns' = conns \setminus \{(from?, to?)\} \\
reqconns' = reqconns \setminus \{(from?, to?)\}
\end{array}$$

Figure 3: The Switch *view*

action is performed; later we shall see how this kind of classification plays a pivotal role.

The operations on a phone constitute one view of the phone 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 a dialling is either *GetBusy* or *GetRing*, but which of the two occurs is not specified in this view.

PhoneSystem

$phones : Id \rightarrowtail Phone$
$Switch$

Bind

$\Delta PhoneSystem$
$\Delta Phone$
$i? : Id$

$\theta phone(i?) = \theta phone \;\land\; \theta phone'(i?) = \theta phone'$

Frame

$\Delta PhoneSystem$
$c? : \mathbb{P}\, Id$

$\forall i \in \mathrm{dom}\, phones \setminus c? \bullet phones(i) = phones'(i)$

$Phones.Initiate = \exists \Delta Phone \bullet Bind \land Phone.Initiate$
$Phones.Dial = \exists \Delta Phone \bullet Bind \land Phone.Dial$
$Phones.GetRing = \exists \Delta Phone \bullet Bind \land Phone.GetRing$
$Phones.GetBusy = \exists \Delta Phone \bullet Bind \land Phone.GetBusy$
$Phones.GetConn = \exists \Delta Phone \bullet Bind \land Phone.GetConn$
$Phones.LoseConn = \exists \Delta Phone \bullet Bind \land Phone.LoseConn$
$Phones.Hangup = \exists \Delta Phone \bullet Bind \land Phone.Hangup$
$Phones.Replace = \exists \Delta Phone \bullet Bind \land Phone.Replace$
$Phones.Answer = \exists \Delta Phone \bullet Bind \land Phone.Answer$
$Phones.Ring = \exists \Delta Phone \bullet Bind \land Phone.Ring$

Figure 4: Composition of Phone *views into single* Phones *view*

The switch that handles connections between phones is specified in another 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). A connection that is active is considered requested until it terminates.

In this view, the dialling of a number by a phone is modelled as 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. It is not executed autonomously by the switch, but corresponds to either party hanging up (a relationship to be expressed later when the views are connected).

$$\begin{array}{l}
\mathit{DialRequest} = \mathit{Switch.Request} \land \mathit{Phones.Dial}[\mathit{from?}/\mathit{i?}] \land \\
\qquad\qquad \mathit{Frame}[\{\mathit{from?}\}/\mathit{c?}] \\
\mathit{RingConnect} = \mathit{Switch.Connect} \land \mathit{Phones.Ring}[\mathit{to?}/\mathit{i?}] \land \\
\qquad \mathit{Phones.GetRing}[\mathit{from?}/\mathit{i?}] \land \mathit{Frame}[\{\mathit{from?}, \mathit{to?}\}/\mathit{c?}] \\
\mathit{RejectBusy} = \mathit{Switch.Reject} \land \mathit{Phones.GetBusy}[\mathit{from?}/\mathit{i?}] \land \\
\qquad\qquad \mathit{Frame}[\{\mathit{from?}\}/\mathit{c?}] \\
\mathit{EndCall} = \mathit{Switch.Terminate} \land \mathit{Frame}[\{\mathit{from?}\}/\mathit{c?}] \land \\
\quad ((\mathit{Phones.Hangup}[\mathit{from?}/\mathit{i?}] \land \mathit{Phones.LoseConn}[\mathit{to?}/\mathit{i?}]) \\
\quad \lor\ (\mathit{Phones.Hangup}[\mathit{to?}/\mathit{i?}] \land \mathit{Phones.LoseConn}[\mathit{from?}/\mathit{i?}])) \\
\mathit{Initiate} = \mathit{Phones.Initiate} \land \mathit{Frame}[\{\mathit{i?}\}/\mathit{c?}] \\
\mathit{Replace} = \mathit{Phones.Replace} \land \mathit{Frame}[\{\mathit{i?}\}/\mathit{c?}] \\
\mathit{Answer} = \mathit{Phones.Answer} \land \mathit{Frame}[\{\mathit{i?}\}/\mathit{c?}]
\end{array}$$

Figure 5: Composition of Phones *view and* Switch *view*

Each phone has its own view. To relate the phone views and the switch view, we first join the phones into a single view (Fig. 4), and then join that view with the switch view (Fig. 5). The global states of the system are given by:

$$\begin{array}{|l}
\hline
\mathit{PhoneSystem} \\
\mathit{phones}: \mathit{Id} \leftrightarrow \mathit{Phone} \\
\mathit{Switch} \\
\hline
\end{array}$$

To extend the scope of an operation defined on a single phone to the whole system, we define a schema that binds the local state of a phone to the corresponding part of the global state

$$\begin{array}{|l}
\hline
\mathit{Bind} \\
\Delta\mathit{PhoneSystem} \\
\Delta\mathit{Phone} \\
\mathit{i?}: \mathit{Id} \\
\hline
\theta\mathit{phone}(\mathit{i?}) = \theta\mathit{phone} \ \land\ \theta\mathit{phone}'(\mathit{i?}) = \theta\mathit{phone}' \\
\hline
\end{array}$$

An operation *op* from the *Phone* view can now be converted to an operation of the *Phones* view, with an argument *i?* identifying the phone on which the operation is executed :

$$\mathit{Phones.op} = \exists \Delta\mathit{Phone} \bullet \mathit{Bind} \land \mathit{Phone.op}$$

This is not just a promotion, since there is no frame condition constraining the status of any other phone. So for example

$$\mathit{Phones.Hangup} = \exists \Delta\mathit{Phone} \bullet \mathit{Bind} \land \mathit{Phone.Hangup}$$

corresponds to replacing the handset of phone *i?*, but the *Bind* schema does not require *phones* to be invariant at $j? \neq i?$. Indeed, the hanging up of one phone will certainly affect the phone it was connected to. To limit the effects of an operation to some set of phones, we define a framing schema that requires all phones with an identifier not in the set $c?$ to be unchanged:

$$\begin{array}{|l}
\textit{Frame} \\
\hline
\Delta \textit{PhoneSystem} \\
c?: \mathbb{P}\, \textit{Id} \\
\hline
\forall i \in \operatorname{dom} \textit{phones} \setminus c? \bullet \textit{phones}(i) = \textit{phones}'(i) \\
\hline
\end{array}$$

On to the final combination: joining the telephones to the switch (Fig. 5). This time there is no inter-view invariant. Instead, each operation of the system as a whole is defined as a combination of operations from the two views. The declaration

$$\begin{aligned}\textit{DialRequest} = \textit{Switch.Request} \wedge \textit{Phones.Dial}[\textit{from}?/i?] \wedge \\ \textit{Frame}[\{\textit{from}?\}/c?]\end{aligned}$$

for example says that the *DialRequest* 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 the switch succeeds in making a connection, for instance, the calling phone hears a ringing tone and the called phone rings:

$$\begin{aligned}&\textit{RingConnect} = \textit{Switch.Connect} \wedge \textit{Phones.Ring}[\textit{to}?/i?] \wedge \\ &\quad \textit{Phones.GetRing}[\textit{from}?/i?] \wedge \textit{Frame}[\{\textit{from}?, \textit{to}?\}/c?]\end{aligned}$$

Finally, we can express the effect of *Hangup*. A call ends when it is terminated in the switch, and one phone hangs up and the other loses the connection:

$$\begin{aligned}&\textit{EndCall} = \textit{Switch.Terminate} \wedge \textit{Frame}[\{\textit{from}?\}/c?] \wedge \\ &\quad ((\textit{Phones.Hangup}[\textit{from}?/i?] \wedge \textit{Phones.LoseConn}[\textit{to}?/i?]) \\ &\quad \vee\ (\textit{Phones.Hangup}[\textit{to}?/i?] \wedge \textit{Phones.LoseConn}[\textit{from}?/i?]))\end{aligned}$$

Again, relating the views involves something like promotion. Promoted operations are disjoint, so that the binding of local state to global state can be combined with a frame condition. Here, however, the binding and framing must be separated. When the local effect of a *Hangup* at some phone is related to the global state, the repercussions on other components of the global state have yet to be established. By conjoining *Hangup*, *LoseConn* and *Terminate*, we can use the state of the switch to determine which phone should lose its connection. It also prevents a *Terminate* from happening when there is no *Hangup*, so the switch cannot drop a call at whim.

View structuring, as before, brings a useful separation of concerns. What tones a phone generates and when a call may be made should not be bound up with questions about how phones are connected and in what order connections are made. Of course views overlap, and eventually the generation of tones and the making of connections must be linked, but this should be delayed as long as possible.

4 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.

Sometimes 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. The subscriber sees the screening list as a set of numbers

$$enemies\colon \mathbb{P}\, Id$$

that is part of the telephone's state, and executes local operations to add and remove numbers from the list such as

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Phone\_Add}} \\
\hline
\Delta Phone \\
i?\colon Id \\
\hline
enemies' = enemies \cup \{i?\} \\
status = dialtone \wedge status' = status \\
\hline
\end{array}$$

The switch, on the other hand, sees the screening lists as a relation between subscribers

$$hates\colon Id \leftrightarrow Id$$

and now a call can be rejected even if the number being called is not busy, so we add a new operation:

$$
\begin{array}{|l}
\textit{Switch.SelectiveReject} \\
\hline
\Delta \textit{Switch} \\
\textit{from}?, \textit{to}?: \textit{Id} \\
\hline
(\textit{from}?, \textit{to}?) \in \textit{reqconns} \cap \textit{hates}^{\sim} \\
\textit{reqconns}' = \textit{reqconns} \setminus \{(\textit{from}?, \textit{to}?)\} \\
\hline
\end{array}
$$

The predicate in the *Connect* operation (Fig. 3) determining when a requested connection can be fulfilled is amended from

$$(\textit{from}?, \textit{to}?) \in \textit{reqconns} \land \textit{to}? \notin (\mathrm{dom}\ \textit{conns} \cup \mathrm{ran}\ \textit{conns})$$

to

$$(\textit{from}?, \textit{to}?) \in \textit{reqconns} \setminus \textit{hates}^{\sim} \land \textit{to}? \notin (\mathrm{dom}\ \textit{conns} \cup \mathrm{ran}\ \textit{conns})$$

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

$$
\begin{array}{|l}
\textit{PhoneSystem} \\
\hline
\textit{phones}: \textit{Id} \rightarrowtail \textit{Phone} \\
\textit{Switch} \\
\hline
\forall i \in \mathrm{dom}\ \textit{phones} \bullet \textit{phones}(i).\textit{enemies} = \textit{hates}[\{i\}] \\
\hline
\end{array}
$$

Conversely, an editor specification is likely to connect operations as well as states. Adding cursor memory (briefly mentioned at the end of 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:

$$op = (V.opNormal \land W_1.opNormal \land \ldots) \lor (V.opError \land W_1.opError \land \ldots)$$

5 Local Reasoning and Operation Entailment

A view is a partial specification of the whole program, in contrast to a module, which is a specification – often complete – of only part of the program. Reading the specification of a single view should thus impart an understanding of all the operations, albeit restricted to some aspect of behaviour. Some operations, however, are specified in only one view, apparently contradicting this aim.

Take the *csrUp* operation of the editor (Fig. 1), for instance. Its specification appears in the *Grid* view, whose state representation makes it easy to talk about the relative position of characters on adjacent lines, and omitted from the *File* view, where its specification would have been tortuous. To allow an independent reading of the *File* view specification, we would like at least all the basic operations to be specified there. Attempting a full specification would be foolish – it was omitted precisely because it was so hard to specify.

But this does not preclude a partial specification. We might, for example, specify *csrUp* in the *File* view as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{File.csrUp}} \\
\hline
\Delta\, \mathit{File} \\
\hline
\exists l : \mathrm{seq}\ \mathit{Char} \bullet \mathit{left} = \mathit{left}' \frown l \ \wedge\ \mathit{right}' = l \frown \mathit{right} \\
\hline
\end{array}$$

This partial description gives only the bare outlines of the behaviour, but includes the crucial features: that the total text of the buffer is unchanged, and that the only effect of the operation is to move the cursor backwards.

How should this specification be treated when the views are composed? We might write

$$\mathit{csrUp} = [\Delta \mathit{Editor} \mid \mathit{Grid.csrUp} \wedge \mathit{File.csrUp}]$$

but this would be entirely equivalent to

$$\mathit{csrUp} = [\Delta \mathit{Editor} \mid \mathit{Grid.csrUp}]$$

since, with the invariant relating the views, the specification of *File.csrUp* could be inferred from *Grid.csrUp*. So instead we write

$$\mathit{csrUp} = [\Delta \mathit{Editor} \mid \mathit{Grid.csrUp}]$$
$$\mathit{csrUp} \vdash [\Delta \mathit{Editor} \mid \mathit{File.csrUp}]$$

The first line defines the system operation. The second asserts that the definition of *File.csrUp* follows logically from it. Instead of constraining the specification further, it signals a proof obligation: to show that the description of the operation given in the *File* view is an approximation to the exact behaviour described in the *Grid* view.

The partial operation is related to the exact operation by simple logical entailment and not refinement. The weakening of the logical formula representing an operation increases its possible effects, and implicitly, the domain in which it may be executed. Refinement, on the other hand, is contravariant: it may weaken the precondition but strengthen the post-condition. In this case, the partial operation allows the cursor to move back to any preceding position in the buffer, or even not to move at all. This imprecision does not compromise the soundness of local reasoning; a property P inferred from *File.csrUp*

$$\mathit{File.csrUp} \vdash P$$

will hold for the exact operation *csrUp*, by the transitivity of $\vdash$. Completeness is lost , of course; not all properties of *csrUp* will be deducible from *File.csrUp*.

6 Why Z?

The choice of Z for our examples was not incidental; it has features that make it especially well suited to view structuring. Their consideration points to the difficulties that must be overcome to apply view-structuring in other languages, such as Larch [GHW85] and VDM [Jon86].

6.1 Frame Conditions

In most specification languages the scope of an operation is given explicitly. Larch, for instance, has a modification clause in addition to pre- and postconditions. 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,. Z, in contrast, has no way to express frame conditions, and the schema would include $y' = y \wedge z' = z$.

For view structuring 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. So when using a language with frame conditions, their scope must be confined to the view in which they appear.

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].

Finally, many of the equalities asserted when views are composed could be dispensed with altogether if a more subtle kind of frame condition were expressible. Instead of saying "only these components change", it would say "only these are altered", allowing others to change only when necessary to maintain invariants. Schuman and Pitt [SP87] have shown how to express this kind of frame condition by defining "completing assertions". Compositionality is retained, but at the expense of introducing meta-level notions.

6.2 Implicit Invariants

The principal novelty of Z is its pervasive use of invariants as a specification mechanism. 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 VDM, on the other hand, invariants play a secondary role, as a proof obligation rather than a specification mechanism. The operations are more explicit, updating every state component that changes. Their is no magic; on the contrary, the specifier must supply all the details and then demonstrate that the resulting specification preserves the invariant.

The VDM approach has two advantages. First, the specification is easier to implement: there tend to be fewer state components and the operation specifications are more operational. Second, the risk of overconstraint (and worse, inconsistency) is lower, the explicit specification being in a sense a constructive proof that post-states exist satisfying the invariant. For designs, these advantages matter a lot, but they seem less crucial in requirements specification.

Z-style invariants, along with conjunction, are the critical mechanism of view structuring. Without them, every operation would have to be fully specified in every view, and the benefit of views would vanish.

The obstacle to view structuring in VDM is style more than semantics. The inter-view invariant might simply be conjoined to the postcondition of each operation, shifting the burden of proof from the maintenance of the invariant (now true by decree) to the implementability of the operation.

6.3 Conjunction and Preconditions

Broadly speaking, pre- and postconditions can be related in three ways. Whether or not they are specified separately, they can be viewed as a single logical formula $pre \wedge post$ (as in Z); as the single formula $pre \Rightarrow post$ (as in Larch and in Hoare's representation of programs as predicates [Hoa85]); or as two distinct formulas (as in VDM). Each interpretation has its benefits. Keeping them apart can simplify the treatment of infinite looping, with the precondition being regarded as a termination criterion. The *pre* $\Rightarrow$ *post* interpretation nicely incorporates the natural contravariance of a procedure, so that the freedom of an implementation I to widen the precondition but narrow the postcondition of a specification S can be expressed as $I \Rightarrow S$.

The Z interpretation is enforced syntactically: one actually writes the specification as a conjunction. This throws a spanner in the works for proving correctness of an implementation, since the contravariance demands that the precondition be retrieved by logical manipulation, and than handled differently from the postcondition.

View composition, however, demands this interpretation. To connect views, we conjoined their operations; for two specifications $pre_1 \wedge post_1$ and $pre_2 \wedge post_2$, this gives*

$$(pre_1 \wedge post_1) \wedge (pre_2 \wedge post_2) = (pre_1 \wedge pre_2) \wedge (post_1 \wedge post_2)$$

and thus the resulting precondition is $(pre_1 \wedge pre_2)$. Each view can thus restrict the circumstances in which an operation may be invoked. A view of a phone switch might prevent a connection to a busy phone, for instance, and a view of an individual phone might prevent a connection when the phone is onhook. Suppose, however, that our specifications were $pre_1 \Rightarrow post_1$ and $pre_2 \Rightarrow post_2$. Their conjunction would be

$$(pre_1 \Rightarrow post_1) \wedge (pre_2 \Rightarrow post_2)$$

which is not equivalent to

$$(pre_1 \wedge pre_2) \Rightarrow (post_1 \wedge post_2)$$

Instead, its precondition is something like $(pre_1 \vee pre_2)$: an operation can happen whenever some view allows it to happen. This does not allow the preconditions of one view to determine the occurrence of operations in another view, and our telephone specification would not work.

The Z interpretation comes from thinking of the specification as a state machine. If an execution of an operation is a transition in the machine, the sentences accepted by the machine are the allowable sequences of states. Ignoring the matching of operation names, the conjunction of two views is then a state machine whose language is the intersection of the languages of the constituent machines.

Unfortunately, the refinement ordering of Z is not consistent with this inter-

* *When the precondition is not given explicitly, this equality becomes an implication*

$$pre(Op_1) \wedge pre(Op_2) \Rightarrow pre(Op_1 \wedge Op_2)$$

but the argument still holds.

pretation. Refinement allows a precondition to be weakened, so that an operation may be executed in more contexts than its specification permits. This is fine if the operations are the procedures of an abstract data type but disastrous if they model the events of a state machine. Weakening the precondition of the *Reject* operation of our phone switch (Fig. 3) would allow the switch to reject a requested connection even when the number being called is not busy! The operation is not a service to the phone user who might benefit from its wider applicability, but an internal action whose more frequent occurrence will not be appreciated. Therefore, if operations are used to model internal actions, the conventional notion of refinement must be abandoned.

Z's precondition is really not a precondition at all, but a firing condition, determining the order in which operations may be invoked. The preconditions of Larch and VDM are disclaimers; they assert that an operation is free to do anything when invoked in a bad state. A specification that uses firing conditions can more easily be interpreted as a process, and thus combined with other process formalisms [Jac88], but becomes confusing if its firing conditions are simultaneously interpreted as preconditions.

7 Related Work

7.1 Views in Z

View structuring can be detected in germinal form 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].

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 representations 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.

This structure is not easily maintained. Consider adding "caller id", which, when a phone rings, causes the calling number to be displayed, and can be disabled at will by the caller. In the view-structured specification (Figs. 2–5), the display would be added as variable in the *Phone* view, along with a bit indicating whether the caller wishes to make her identity known, and a local toggling operation. The functionality is then expressed by an inter-view invariant that says that if a phone is ringing (*Phone* view) and connected to another phone (*Switch* view) that has its bit set to true (*Phone* view), then the display of the first phone holds the number of the second. But the elaboration has no natural home in Woodcock's specification, requiring ad hoc changes to the global state and operations.

7.2 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, can 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.

The idea of recording a proof obligation to relate the partial and full specification of an operation is taken from the Larch Shared Language, which provides an "implies" clause in which properties of operators that are expected to follow from their definition are asserted.

7.3 Viewpoints in Requirements Analysis

The need for multiple views (usually called "viewpoints") in requirements analysis has long been recognized, but most of the work in this area starts from assumptions that are different to mine, and has a different purpose altogether.

The viewpoints of Finkelstein, Kramer et al [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. But the risk of inconsistency is a problem. This approach dispenses with any kind of universal model in which connections could be interpreted, advocating an operational mechanism instead. Consistency is enforced by a set of actions that are triggered by local changes to a viewpoint and propagated to other viewpoints. Viewpoint construction in this approach is a kind of cooperative work (with problems similar to the joint editing of any document) whose distributed nature is a fundamental assumption.

For Wallis [Wal92, A&93], viewpoints are partial specifications of functionality, written in Z but by different people, to be reconciled as soon as possible. First, any irreconcilable differences must be eradicated; then the viewpoints must be amalgamated. The process of reconciliation and amalgamation is the focus of this work, and not the structure of the initial specifications (which is standard). The division into viewpoints does not arise from any fundamental difficulty in writing a single specification, but rather because different people with different aims wrote different versions of the specification. By analogy to programming, view structuring is about a single program; Wallis's concern is the problem of integrating different versions [HPR89].

For view structuring, the same reasons that favour initial separation of views militate against their amalgamation, and, in their present form, Wallis's techniques do not seem strong enough to handle views whose state components are not matched in a fairly simple manner.

7.4 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.)

7.5 Structuring Mechanisms Similar to Views: Descriptions

Zave and Jackson propose a structuring mechanism for specifications whose units they call "descriptions" [ZJ93]. In two respects their work is more ambitious than mine. First, descriptions can separate more fundamental concerns than aspects of functionality, distinguishing the specification of the machine, the specification of its domain, and the specification of the environment in which the machine operates [Jac94]. In contrast, the views of this paper are all descriptions of machines.

Second, descriptions can be cast in different paradigms. A telephone specification might use Z to describe data aspects and JSP to describe the ordering of events [ZJ93]. To interpret these diverse descriptions as a single, coherent specification, Zave and Jackson abandon their standard semantics and propose instead a translation into a minimalist logical theory. The primitive predicates of this theory are explicitly "designated" by the specifier, who says, for example, that *isRing(e, t, n)* means that "in event *e* at time *t* telephone *n* rings". The entire specification can thus be translated, at least in principle, to a statement similar to a scientific claim that can be refuted by observation.

The synchronization of operations in my telephone specification can be seen as a special case of "event classification" [ZJ91, ZJ94]. A description can partition an event class into subclasses; other descriptions may then refer to the names of the subclasses. Replacing the handset of a phone, for example, might be an event called *Replace* that some description classifies into *Hangup* (when terminating a call) and *EndSession* (when no call is in progress). Another description (of billing, say) might then refer only to *Hangup* events.

The choice of *GetBusy* or *GetRing*, in the *Phone* view (Fig. 2) is non-deterministic, but resolved by conjoining them to *Reject* or *Connect* whose choice in the *Switch* view (Fig. 3) is fully determined. This is a weak form of event classification. In Jackson and Zave's examples, the description that resolves the choice exports the names of the operations, so the classification is localized. The standard semantics of Z cannot accommodate this, since operations are just relations on states whose names have no significance. Interpreting the conjunction of two Z operations as synchronization is a convenient intuition, but technically it makes no sense to talk of operations occurring in one or other view.

8 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 be confined within a view.

Z is especially well-suited to view structuring. The vital features are: the relationship between pre- and postconditions; the lack of implicit frame conditions; and the ability to conjoin operations and assert global invariants.

Despite its benefits, view-structuring has yet to be fully exploited. Although its seeds are 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

Many thanks to Gregory Abowd, Michael Jackson, Jeannette Wing, Michal Young and Pamela Zave, who gave me helpful comments on drafts of this paper. I am also grateful to David Garlan for his insights into Z, and to Mary Shaw, whose advice has been invaluable.

References

[A&93] M. Ainsworth, A.H. Cruickshank, L.J. Groves and P.J.L. Wallis, "Formal Specification via Viewpoints", *Proc. 13th New Zealand Computer Conference*, New Zealand Computer Society, Auckland, New Zealand, 1993.

[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.

[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 *Specification Case Studies*, Ian Hayes, Prentice Hall International, 2nd ed., 1993.

[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.

[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.

[HPR89] S. Horwitz, J. Prins and T. Reps, "Integrating Non-interfering Versions of Programs", *ACM Trans. on Programming Languages and Systems*, 11(3), Jan. 1990.

[Jac88] Daniel Jackson, *Composing Data and Process Descriptions in the Design of Software Systems*, Technical Report MIT/LCS/TR-419, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, May 1988.

[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.

[Jon86] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International, 1986.

[JZ93] Michael Jackson and Pamela Zave, "Domain Descriptions", *Proc. IEEE International Conference on Requirements Engineering*, IEEE Computer Society Press, pp. 56–64, 1993.

[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.

[SP87] S.A. Schuman and D.H. Pitt, "Object-Oriented Subsystem Specification", in *Program Specification and Transformation*, ed. L.G.L.T. Meertens, North Holland, pp. 313–341, 1987.

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

[Wal92] Peter J.L. Wallis, *A New Approach to Modular Formal Description*, Technical Report 92-57, University of Bath, 1992.

[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.

[ZJ91] Pamela Zave and Michael Jackson, "Techniques for Partial Specification of Switching Systems", in *VDM'91: Formal Software Development Methods, Proc. 4th International Symposium of VDM Europe,* Springer-Verlag, pp. 511–525, 1991.

[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", submitted for publication, Feb. 1994.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890