Chapter 4

Specification of the Unix filing system

Carroll Morgan and Bernard Sufrin

Abstract A specification of the Unix filing system is given using a notation based on elementary mathematical set theory. The notation used involves very few special constructs of its own.

The specification is detailed enough to capture the filing system's behaviour at the system call level, yet abstracts from issues of data representation, whether within programs or on the storage medium, and from the description of any algorithms which might be used to implement the system.

The presentation of the specification is in several stages, each new stage building on its predecessors; major concepts are introduced separately so that they may be easily understood. The notation used allows these separate stages to be joined together to give a complete description of each filing system operation – including its error conditions.

4.1 Introduction

The Unix [52] operating system is widely known, and its filing system is well understood. Why, then, do we present a formal specification of it here? It is because the idea of formalising the specification of computer-based systems has yet to receive widespread acceptance among computing practitioners, and in our view this is because very few realistic examples have been published. Publishing a *post hoc* specification of aspects of the Unix filestore offered us the possibility of showing how to use a mathematically based notation to capture important aspects of the behaviour of a system that is clearly not just a toy.

The use of natural language – *without* supporting mathematics – has serious limitations as a vehicle for the description of computer systems. As anyone who has ever used an operating system will confirm, the manuals cannot tell the whole story about the behaviour of a system. Indeed, almost every programmer who starts to use a new operating system sets up a number of *experiments*, by which she attempts to discover how it 'really' behaves. It is a commonplace observation that large computer systems, operating systems in particular, accumulate around themselves a body of folklore – necessary knowledge for anybody who wishes to use them effectively – and a number of 'gurus' – people who understand the hidden secrets of the system because they have read . . . the source code!

In our approach to the description of computer systems we use natural language *together with* the formal language of mathematics. And our particular style is simply a means of presenting the formal part of the description in a way that can be easily manipulated and understood. The formal descriptions themselves are given in elementary mathematical set theory, which is convenient for this purpose because programs are themselves mathematical objects [1, 20]. The difference between a mathematical specification and a program is only of degree: they are objects drawn from the same continuum. This uniformity allows, for example, the refinement of formal specifications into programs to be mathematically verified [33].

By using a mixture of natural language and elementary set theory we have enabled ourselves to give a description which is comprehensive enough to describe the essential aspects of the system's behaviour, but is sufficiently abstract that it will not burden the reader with the kind of detail that appears in the source code. In particular, it has allowed us to avoid describing the representation of data on external media and within programs and to refrain from presenting details of the algorithms that are used to implement the filestore operations. Thus the specification here might occur midway along the path from a more abstract but informal specification – a description such as is given in [52] – to a more concrete one – the source code itself [38]. This intermediate level of abstraction is one which conveniently captures the behaviour of the system at the system call level, without being concerned with representational matters.

At each stage of presentation, the static (invariant) properties of the system are characterised by *naming* the observations that can be made of it, attributing a (set theoretical) *type* to each observation, and recording the invariant relationships between these observations as a collection of predicates.

The dynamic behaviour of the system is characterised by giving – for each of the operations under which the system evolves – the names of the observations that can be made *before* the operation, the names of those that can be made *after* the operation, and a collection of predicates that relate these two sets of observations. The operations in question in this case are just the Unix system calls, and the observations we are interested in may include components of the system state, and the 'arguments' and 'results' of system calls.

When providing a specification (such as this one) which is a 'tutorial' exercise rather than a reference manual, the concepts must be introduced gradually so as not to overwhelm the reader with immediate detail. The specification begins with the definition of a file alone, but ultimately includes channels (file descriptors), file identifiers (i-numbers), and even the abstract format of a directory file. Error conditions are treated last of all, so that they do not complicate the description of what usually happens with the problems of what might happen.

One novel aspect of the specification style is the use of a *homogeneous* framework – *schemas* – to characterise both dynamic and static properties. Schemas supplement the notation of set theory by providing notations for naming and combining groups of observations and predicates, and methods of reasoning about the combinations; this is exactly what is needed to present the specification gradually. Moreover, since the tutorial style of the specification is based on mathematics, it is necessary when providing a reference manual only to collect its definitions into one unit – a summary, in effect – using the laws of combination of schemas.

The value of a specification such as this is that it defines the system in question, so that its properties may be determined by reasoning rather than by performing experiments on the system itself – these could be difficult (if the system is complex) and costly (if it has not yet been built). Since several specifications can be constructed for one system, each may take a point of view, or adopt a level of abstraction, which is appropriate to the questions it is required to answer. And if these specifications are presented within a formal framework, the question of their meaning and consistency is only a mathematical one, and so can be answered by mathematical means rather than by armwaving. But of course the *real* payoff is that when the system is built and in use, all those painful – and perplexing – visits to the guru can be avoided.

4.2 Scope of the specification

The system described is Unix Level 6. The operations covered include the system calls

read	write	create
seek	open	close
fstat	link	unlink

and the commands

ls	move

Some of the features not treated are

- special files;
- pipes; and
- file access permissions.

Some of the more practical considerations, such as storage device size, are examined in Appendix 4.5. The treatment of errors covers only a few examples, but illustrates the technique which would apply to them all.

4.3 The specification

4.3.1 Bytes and files

The ultimate constituent of the filing system is the *byte*; the set of all bytes is called $BYTE$:

$$BYTE == 0 \,..\, 255$$

A *file* is a finite *sequence* of bytes of any length[1] (including the null sequence $\langle\ \rangle$ of length 0):

$$FILE == \operatorname{seq} BYTE$$

[1] See Appendix 4.5.1.

In general, a sequence of X is a partial function from the natural numbers ($\mathbb{N}$) into X; for any sequence s and natural number n, $s(n)$ is the nth element of s (if defined). Thus for any f of type $FILE$ $f(1)$ is the first byte of the file. The function $\#$ gives the length of any sequence; hence $\#f$ is the size of the file, and $f(\#f)$ is its last byte.

4.3.2 Reading and writing

When a file is read the file itself is not changed; if $file'$ is the file's value after the operation, and $file$ is its value before, then

$$file' = file$$

The result of *reading* a file is a sequence $data!$ of bytes:

$$data! : \mathrm{seq}\, BYTE$$

The value of $data!$ is determined by an offset into the file and a length to be read; both are natural numbers (i.e. non-negative integers):

$$offset?, length? : \mathbb{N}$$

and in fact

$$data! = (1 \mathrel{..} length?) \lhd (file \mathrel{\mathrm{after}} offset?)$$

The infix operator 'after' takes a sequence, in this case *file*, as its first argument and an offset as its second argument, and returns the subsequence of *file* beginning after the offset. The first *length*? bytes (if there are that many) are then selected from the resultant sequence to give the data returned by the read. The operator 'after' has the following definition:

$$
\begin{array}{l}
[X] \\
\hline
\_ \mathrel{\mathrm{after}} \_ : \mathrm{seq}\, X \times \mathbb{N} \to \mathrm{seq}\, X \\
\hline
\forall\, s : \mathrm{seq}\, X;\ offset : \mathbb{N} \bullet \mathrm{dom}(s \mathrel{\mathrm{after}} offset) = (1 \mathrel{..} \#s - offset) \land \\
(\forall\, n : \mathbb{N} \bullet \\
\qquad (n + offset) \in \mathrm{dom}\, s \Rightarrow \\
\qquad\qquad (s \mathrel{\mathrm{after}} offset)(n) = s(n + offset))
\end{array}
$$

Therefore

$$(file \mathrel{\mathrm{after}} offset?)$$

is a formalisation of

file after the first *offset?* bytes

This means that the first byte of the file has offset 0.

The domain restriction operator ($\lhd$) here excludes any element whose index is not in the set $1 \mathrel{..} length?$; $data!$ is therefore

file, after *offset*?, for no more than *length*?

For example, if

$$\begin{aligned} \mathit{file} &= \langle X, A, N, F, R, E, D \rangle \\ \mathit{offset?} &= 2 \\ \mathit{length?} &= 3 \end{aligned}$$

then

$$(\mathit{file} \text{ after } 2)(n) = \mathit{file}(n+2)$$

That is,

$$\mathit{file} \text{ after } \mathit{offset?} = \langle N, F, R, E, D \rangle$$

and therefore

$$\mathit{data!} = (1 \mathinner{..} 3) \lhd \langle N, F, R, E, D \rangle = \langle N, F, R \rangle$$

All of these properties may be collected in a *schema* which defines the reading operation:

$$\begin{array}{|l}\hline \mathit{file}, \mathit{file}' : \mathit{FILE} \\ \mathit{offset?}, \mathit{length?} : \mathbb{N} \\ \mathit{data!} : \operatorname{seq} \mathit{BYTE} \\ \hline \mathit{file}' = \mathit{file} \\ \mathit{data!} = (1 \mathinner{..} \mathit{length?}) \lhd (\mathit{file} \text{ after } \mathit{offset?}) \\ \hline \end{array}$$

When a schema is used (as it is here) to characterise an operation, its *signature*

$$\begin{array}{l} \mathit{file}, \mathit{file}' : \mathit{FILE} \\ \mathit{offset?}, \mathit{length?} : \mathbb{N} \\ \mathit{data!} : \operatorname{seq} \mathit{BYTE} \end{array}$$

gives names and types to the observations that can be made before and after the operation. The *predicate*

$$\begin{array}{l} \mathit{file}' = \mathit{file} \\ \mathit{data!} = (1 \mathinner{..} \mathit{length?}) \lhd (\mathit{file} \text{ after } \mathit{offset?}) \end{array}$$

relates these observations to one another.

Naming a schema allows it to be referred to within subsequent definitions; the name is written as part of the enclosing 'box'.

$$\begin{array}{|l}\hline \mathit{readFILE} \\ \hline \mathit{file}, \mathit{file}' : \mathit{FILE} \\ \mathit{offset?}, \mathit{length?} : \mathbb{N} \\ \mathit{data!} : \operatorname{seq} \mathit{BYTE} \\ \hline \mathit{file}' = \mathit{file} \\ \mathit{data!} = (1 \mathinner{..} \mathit{length?}) \lhd (\mathit{file} \text{ after } \mathit{offset?}) \\ \hline \end{array}$$

The definition above can be read:

> The *readFILE* operation does not change the file. It expects an offset and length as parameters, and returns as its result the data read. The value returned is the longest sequence of bytes, of length not greater than that requested, which begins at the given offset in the file.

To define the *writeFILE* operation, a similar schema is used; this time, however, the file *is* changed.

The byte *ZERO* is used in the definition of *writeFILE*; it is a distinguished element of $BYTE$:

$$ZERO == 0$$

And *zero(k)* is a sequence of length k containing only *ZERO* bytes:

$$\begin{array}{|l}
zero : \mathbb{N} \to \mathrm{seq}\, BYTE \\
\hline
\forall n : \mathbb{N} \bullet zero(n) = (\lambda\, k : 1 \mathinner{..} n \bullet ZERO)
\end{array}$$

Writing with an offset greater than the file length leaves *ZERO* bytes between the previous end of the file and the newly written data.

$$\begin{array}{|l|}
\hline
\textit{writeFILE} \\
\hline
file, file' : FILE \\
offset? : \mathbb{N} \\
data? : \mathrm{seq}\, BYTE \\
\hline
file' = zero(offset?) \oplus file \oplus (data? \mathbin{\mathrm{shift}} offset?) \\
\hline
\end{array}$$

The infix operator 'shift' takes a sequence, in this case $data?$ and an offset and shifts $data?$ by the offset. It has the following definition:

$$\begin{array}{|l}
[X] \\
\hline\hline
_ \mathbin{\mathrm{shift}} _ : \mathrm{seq}\, X \times \mathbb{N} \to (\mathbb{N} \pfun X) \\
\hline
\forall s : \mathrm{seq}\, X;\ offset : \mathbb{N} \bullet \\
\mathrm{dom}(s \mathbin{\mathrm{shift}} offset) = \{ i : \mathrm{dom}\, s \bullet i + offset \} \land \\
(\forall n : \mathrm{dom}(s \mathbin{\mathrm{shift}} offset) \bullet \\
\quad (s \mathbin{\mathrm{shift}} offset)(n) = s(n - offset)) \\
\hline
\end{array}$$

'$\oplus$' is the function overriding operator: $f \oplus g$ behaves like g except where g is undefined, in which case it behaves like f. Thus the value of any byte in the file

$$zero(offset?) \oplus file \oplus (data? \mathbin{\mathrm{shift}} offset?)$$

is determined first by the written $data?$, then by the previous contents of the *file*, and finally is *ZERO* otherwise. The length of the new file is

$$max(\#file, offset? + \#data?)$$

Thus

$$\begin{aligned}
file &= \langle X, A, N, F, R, E, D \rangle \land \\
offset? &= 8 \land
\end{aligned}$$

$$
\begin{aligned}
&& data? &= \langle N, U, N, I, B, A, D \rangle \\
&\Rightarrow && \\
&& file' &= \langle \sqcup, \sqcup, \sqcup, \sqcup, \sqcup, \sqcup, \sqcup, \sqcup \rangle \oplus \langle X, A, N, F, R, E, D \rangle \oplus \\
&& &\qquad (\langle N, U, N, I, B, A, D \rangle \text{ shift } 8) \\
&\Rightarrow && \\
&& file' &= \langle X, A, N, F, R, E, D, \sqcup \rangle \oplus (\langle N, U, N, I, B, A, D \rangle \text{ shift } 8) \\
&\Rightarrow && \\
&& file' &= \langle X, A, N, F, R, E, D, \sqcup, N, U, N, I, B, A, D \rangle
\end{aligned}
$$

(The byte *ZERO* is here represented by a '$\sqcup$'.)

A consequence of this definition is that *writeFILE* is possible for *all* values of *file*, *offset*?, and *data*? (subject to any limitation on the maximum size of files in general); formally, this is shown by proving that there is always a value for $file'$, consistent with its type *FILE* ($\text{seq}\, BYTE$), such that the following predicate holds:

$$file' = zero(offset?) \oplus file \oplus (data? \text{ shift } offset?)$$

4.3.3 File storage

The *file storage* system allows files to be stored and retrieved using *file identifiers*; the set of all file identifiers is called *FID*:

$$[FID]$$

The storage system is characterised by a single observation: a partial function[2] from *FID* to *FILE*.

$$
\begin{array}{|l}
\hline
\textit{SS} \\
\hline
fstore : FID \nrightarrow FILE \\
\hline
\end{array}
$$

An empty file may be *created* in the storage system by supplying its identifier as a parameter to an operation which changes an *old* storage system, *SS*, into a *new* one which contains the created file, SS'. *SS* is equivalent to

$$fstore : FID \nrightarrow FILE$$

so SS' is equivalent to

$$fstore' : FID \nrightarrow FILE$$

Thus, the effect of decorating a schema name is to decorate the names of its observation(s).

The operation that creates an empty file is defined by the schema

$$
\begin{array}{|l}
\hline
\textit{createSS} \\
\hline
SS \\
SS' \\
fid : FID \\
\hline
fstore' = fstore \oplus \{ fid \mapsto \langle\, \rangle \} \\
\hline
\end{array}
$$

[2]See Appendices 4.5.2 and 4.5.3.

The new store $fstore'$ is identical to the old except that fid now refers to the empty file $\langle\ \rangle$– *whether or not it referred to a file previously.* Thus creating an existing file empties it. We do not write '$fid?$' because later it will be seen that these file identifiers are in fact not visible to the user.

Destroying a file is defined

$$
\begin{array}{|l}
\hline
\textit{destroySS} \\
SS \\
SS' \\
fid : FID \\
\hline
fid \in \operatorname{dom} fstore \\
fstore' = \{fid\} \mathbin{⩤} fstore \\
\hline
\end{array}
$$

Naturally, a file must exist ($\in \operatorname{dom} fstore$) to be destroyed. The new $fstore'$ is identical to the old except that there is no file referred to by fid:

$$fid \notin \operatorname{dom} fstore'$$

4.3.4 Reading and writing stored files – framing

Reading a *stored* file is defined by the following schema:

$$
\begin{array}{|l}
\hline
SS \\
SS' \\
fid : FID \\
offset?, length? : \mathbb{N} \\
data! : \operatorname{seq} BYTE \\
file, file' : FILE \\
\hline
fid \in \operatorname{dom} fstore \\
file = fstore(fid) \\
data! = (1 \mathrel{..} length?) \lhd (file \text{ after } offset?) \\
file' = file \\
fstore' = fstore \oplus \{fid \mapsto file'\} \\
\hline
\end{array}
$$

The file read is that referred to by fid, the data output is from $offset?$ for $length?$ (as before), and the file is not changed.

This long-winded definition of reading a stored file shows that it is in fact a combination of the definitions given above for

- reading a file (*readFILE*); and
- the storage system (*SS*).

This kind of combination is called *framing*, because it involves specifying

- *which* file is read or written; and
- that the *other* files are unaffected.

That is, a frame is supplied within which the operation occurs. The following schema states this framing combination generally:

$$
\begin{array}{|l}
\hline
\Phi SS \\
\hline
SS \\
SS' \\
\mathit{file}, \mathit{file}' : \mathit{FILE} \\
\mathit{fid} : \mathit{FID} \\
\hline
\mathit{fid} \in \operatorname{dom} \mathit{fstore} \\
\mathit{file} = \mathit{fstore}(\mathit{fid}) \\
\mathit{fstore}' = \mathit{fstore} \oplus \{\mathit{fid} \mapsto \mathit{file}'\} \\
\hline
\end{array}
$$

fid denotes the file affected in fstore – namely $(\mathit{file}, \mathit{file}')$ – and no other file is changed. Φ is conventionally used as the first letter of framing schemas (Φ for *frame*).

Although the definition given above of reading a stored file could have stated explicitly that the filestore is not changed – $\mathit{fstore}' = \mathit{fstore}$ – this is really a *consequence* of the fact that the file itself is not changed. And the framing schema ΦSS makes it much easier to write such definitions generally – for example, the operation above could be defined as follows:

$$
\begin{array}{|l}
\hline
\mathit{readSS} \\
\hline
\Phi SS \\
\mathit{readFILE} \\
\hline
\end{array}
$$

The signatures and predicates of the two schemas are combined separately and then joined to form the new schema. Where the two schemas *share* a named observation in their signatures, it appears only once in the new schema. Thus, although file and file' occur in both $\mathit{readFILE}$ and ΦSS, they appear only once in readSS.

Writing a stored file is defined similarly.

$$
\begin{array}{|l}
\hline
\mathit{writeSS} \\
\hline
\Phi SS \\
\mathit{writeFILE} \\
\hline
\end{array}
$$

Its definition may be expanded:

$$
\begin{array}{|l}
\hline
SS \\
SS' \\
\mathit{fid} : \mathit{FID} \\
\mathit{offset}? : \mathbb{N} \\
\mathit{data}? : \operatorname{seq} \mathit{BYTE} \\
\mathit{file}, \mathit{file}' : \mathit{FILE} \\
\hline
\mathit{fid} \in \operatorname{dom} \mathit{fstore} \\
\mathit{file} = \mathit{fstore}(\mathit{fid}) \\
\mathit{file}' = \mathit{zero}(\mathit{offset}?) \oplus \mathit{file} \oplus (\mathit{data}? \operatorname{shift} \mathit{offset}?) \\
\mathit{fstore}' = \mathit{fstore} \oplus \{\mathit{fid} \mapsto \mathit{file}'\} \\
\hline
\end{array}
$$

As in readSS, file and file' appear only once in this combination.

4.3.5 Hiding and simplification

In the schema *readSS* the observations *file* and *file′* are entirely determined in value by the other observations of the schema. Unless it is necessary to observe the *whole file* involved in a read or write operation, these observations have become inessential to the specification. Observations such as these are called *auxiliary*.

Hiding auxiliary observations can allow simplification of the schema in which they occur. Components are hidden by removing them from the signature of the schema and by existentially quantifying them in the predicate part. *readSS*, with *file* and *file′* hidden, is written $readSS \setminus (file, file')$ and is in full

$$
\begin{array}{|l}
SS \\
SS' \\
fid : FID \\
offset?, length? : \mathbb{N} \\
data! : \operatorname{seq} BYTE \\
\hline
(\exists file, file' : FILE \bullet \\
\quad fid \in \operatorname{dom} fstore \\
\quad file = fstore(fid) \\
\quad data! = (1 \mathrel{..} length?) \lhd (file \text{ after } offset?) \\
\quad file' = file \\
\quad fstore' = fstore \oplus \{fid \mapsto file'\})
\end{array}
$$

This schema can be simplified using basic predicate calculus:

$$
\begin{array}{|l}
SS \\
SS' \\
fid : FID \\
offset?, length? : \mathbb{N} \\
data! : \operatorname{seq} BYTE \\
\hline
fid \in \operatorname{dom} fstore \\
fstore' = fstore \\
data! = (1 \mathrel{..} length?) \lhd (fstore(fid) \text{ after } offset?)
\end{array}
$$

Writing may be treated similarly.

4.3.6 Sequential access to files

The read and write operations described so far support random access; in order to allow easy sequential use of these operations, a *channel* is defined which remembers the current position in the file.

$$
\begin{array}{|l}
\multicolumn{1}{l}{CHAN} \\
fid : FID \\
posn : \mathbb{N}
\end{array}
$$

A channel has a file identifier fid – which may refer to a file in $fstore$ – and a position $posn$ within the file. As usual, operations involving the channel take the form of a predicate relating the observations of

$$CHAN$$

to those of

$$CHAN'$$

They have the additional property that the fid of a channel is never changed. The schema $\Delta CHAN$ expresses the general properties of any operation on a channel (Δ for *change*).

$$\begin{array}{|l}
\multicolumn{1}{l}{\Delta CHAN} \\
\hline
CHAN \\
CHAN' \\
\hline
fid' = fid \\
\hline
\end{array}$$

Sequential reading and writing using channels is easily characterised by combining the previous definitions.

$$\begin{array}{|l}
\multicolumn{1}{l}{readCHAN} \\
\hline
readSS \\
\Delta CHAN \\
\hline
offset? = posn \\
posn' = posn + \#data! \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{writeCHAN} \\
\hline
writeSS \\
\Delta CHAN \\
\hline
offset? = posn \\
posn' = posn + \#data? \\
\hline
\end{array}$$

In addition, there is an operation $seekCHAN$ which changes only the position.[3]

$$\begin{array}{|l}
\multicolumn{1}{l}{seekCHAN} \\
\hline
SS \\
SS' \\
\Delta CHAN \\
newposn? : \mathbb{N} \\
\hline
fstore' = fstore \\
posn' = newposn? \\
\hline
\end{array}$$

The new position is not constrained to be within the file.[4]

[3] See Appendix 4.5.4.
[4] See Appendix 4.5.5.

4.3.7 Channel system

A *channel storage* system may be defined which is analogous to the file storage system; it allows channels to be stored and retrieved using channel identifiers taken from the set *CID*. A channel identifier is a Unix ‘file descriptor’:

$[CID]$

$$
\begin{array}{|l}
\textit{CS} \\
\hline
\textit{cstore} : \textit{CID} \pfun \textit{CHAN} \\
\hline
\end{array}
$$

Operations on the channel system have the general form

$$
\begin{array}{|l}
\Delta \textit{CS} \\
\hline
\textit{CS} \\
\textit{CS}' \\
\hline
\end{array}
$$

These operations are defined below:

$$
\begin{array}{|l}
\textit{openCS} \\
\hline
\Delta \textit{CS} \\
\textit{CHAN} \\
\textit{cid}! : \textit{CID} \\
\hline
\textit{cid}! \notin \operatorname{dom} \textit{cstore} \\
\textit{posn} = 0 \\
\textit{cstore}' = \textit{cstore} \oplus \{\textit{cid}! \mapsto \theta \textit{CHAN}\} \\
\hline
\end{array}
$$

openCS creates a new channel and returns a new identifier which refers to it; the new channel’s position is zero. $\theta\textit{CHAN}$ stands for the ‘pair’ with components *posn* and *fid*. In this case the component *posn* is zero and the component *fid* is unconstrained (its value will be determined at a later stage).

$$
\begin{array}{|l}
\textit{closeCS} \\
\hline
\Delta \textit{CS} \\
\textit{cid}? : \textit{CID} \\
\hline
\textit{cid}? \in \operatorname{dom} \textit{cstore} \\
\textit{cstore}' = \{\textit{cid}?\} \ndres \textit{cstore} \\
\hline
\end{array}
$$

closeCS removes a channel from the channel system.

4.3.8 The access system

The storage and channel systems together form the *access* system.

$$
\begin{array}{|l}
\textit{AS} \\
\hline
\textit{SS} \\
\textit{CS} \\
\hline
\{\textit{chan} : \operatorname{ran} \textit{cstore} \bullet \textit{chan}.\textit{fid}\} \subseteq \operatorname{dom} \textit{fstore} \\
\hline
\end{array}
$$

The predicate in the above schema requires that every channel must refer to an existing file. This property is an *invariant* of the access system and is preserved by all operations on it. The schema ΔAS automatically includes the invariant of both the initial (AS) and final (AS') state.

$$\begin{array}{|l}
\hline
\Delta AS \\
\hline
AS \\
AS' \\
\hline
\end{array}$$

Reading, writing and seeking in the access subsystem are defined with the assistance of a framing schema.

$$\begin{array}{|l}
\hline
\Phi AS \\
\hline
\Delta AS \\
\Delta CHAN \\
cid? : CID \\
\hline
cid? \in \mathrm{dom}\ cstore \\
\theta CHAN = cstore(cid?) \\
cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\} \\
\hline
\end{array}$$

$\theta CHAN$ in the predicate part is the channel with components *fid* and *posn* as they appear in $\Delta CHAN$; $\theta CHAN'$ is similar but with components fid' and $posn'$.

Reading, writing and seeking in the access system are now defined by combination of previous definitions and the framing schema ΦAS; as usual, some auxiliary variables will be hidden.

The operator $\wedge$ when applied to two schemas is shorthand for writing the two together; that is,

$$\Phi AS \wedge readCHAN$$

is just

$$\begin{array}{|l}
\hline
\Phi AS \\
readCHAN \\
\hline
\end{array}$$

The definitions are

$$\begin{array}{rcl}
readAS & \widehat{=} & (\Phi AS \wedge readCHAN) \setminus (offset?, fid', posn', file') \\
writeAS & \widehat{=} & (\Phi AS \wedge writeCHAN) \setminus (offset?, fid', posn') \\
seekAS & \widehat{=} & (\Phi AS \wedge seekCHAN) \setminus (fid, fid', posn, posn')
\end{array}$$

which when expanded and simplified give

$$
\begin{array}{|l}
\hline
\textit{readAS} \\
\Delta AS \\
cid? : CID \\
length? : \mathbb{N} \\
data! : \mathrm{seq}\, BYTE \\
CHAN \\
file : FILE \\
\hline
cid? \in \mathrm{dom}\, cstore \\
\theta CHAN = cstore(cid?) \\
file = fstore(fid) \\
fstore' = fstore \\
(\exists\, CHAN' \bullet posn' = posn + \#data! \land \\
\quad fid' = fid \land \\
\quad cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\}) \land \\
data! = (1 \mathinner{..} length?) \lhd (file \text{ after } posn) \\
\hline
\end{array}
$$

and

$$
\begin{array}{|l}
\hline
\textit{writeAS} \\
\Delta AS \\
cid? : CID \\
data? : \mathrm{seq}\, BYTE \\
CHAN \\
file, file' : FILE \\
\hline
cid? \in \mathrm{dom}\, cstore \\
\theta CHAN = cstore(cid?) \\
file = fstore(fid) \\
file' = zero(posn) \oplus file \oplus (data? \text{ shift } posn) \\
fstore' = fstore \oplus \{fid \mapsto file'\} \\
(\exists\, CHAN' \bullet posn' = posn + \#data? \land \\
\quad fid' = fid \land \\
\quad cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\}) \\
\hline
\end{array}
$$

and

$$
\begin{array}{|l}
\hline
\textit{seekAS} \\
\Delta AS \\
cid? : CID \\
newposn? : \mathbb{N} \\
\hline
cid? \in \mathrm{dom}\, cstore \\
fstore' = fstore \\
(\exists\, CHAN' \bullet posn' = newposn? \land \\
\quad fid' = (cstore\ cid?).fid \land \\
\quad cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\}) \\
\hline
\end{array}
$$

In addition to the three operations above, the *fstat* operation, which returns the size of the file accessed with a given *CID*, can be defined by

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{fstat}}\\ \hline
\Delta AS\\
cid? : CID\\
size! : \mathbb{N}\\ \hline
cid? \in \operatorname{dom} cstore\\
fstore' = fstore\\
cstore' = cstore\\
size! = \#(fstore((cstore\ cid?).fid))\\ \hline
\end{array}
$$

4.3.9 A file naming system

The naming system associates file names from the set *NAME* with file identifiers *FID*; these file names will normally be chosen by the users of the file system.

$$
\begin{array}{|l}
\multicolumn{1}{l}{NS0}\\ \hline
nstore : NAME \rightarrowtail\!\!\!\!\!\!+ FID\\ \hline
\end{array}
$$

To create an association in the naming system, a *name* and *fid* are supplied; the new association *overrides any existing association for that name*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{createNS}}\\ \hline
\Delta NS0\\
name? : NAME\\
fid : FID\\ \hline
nstore' = nstore \oplus \{name? \mapsto fid\}\\ \hline
\end{array}
$$

Given a *name*, its *fid* may be discovered.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{lookupNS}}\\ \hline
\Xi NS0\\
name? : NAME\\
fid' : FID\\ \hline
name? \in \operatorname{dom} nstore\\
fid' = nstore(name?)\\ \hline
\end{array}
$$

The schema $\Xi NS0$ expresses the observation that the naming system is unaffected; its definition is

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Xi NS0}\\ \hline
NS0\\
NS0'\\ \hline
nstore' = nstore\\ \hline
\end{array}
$$

ΞCS and ΞSS are defined similarly.

Finally, given a *name*?, any association it has may be destroyed (this is the *unlink* operation).

$$
\begin{array}{|l}
\hline
\textit{destroyNS} \\
\Delta NS0 \\
name? : NAME \\
\hline
name? \in \mathrm{dom}\, nstore \\
nstore' = \{name?\} \lhd nstore \\
\hline
\end{array}
$$

4.3.10 Pathnames and directories

By further revealing file names to be sequences of *syllables*

$$[SYL]$$
$$NAME == \mathrm{seq}\, SYL$$

it is possible to provide more structure in the name space as a whole (the name space is dom *nstore*). The naming system is augmented by a set of *directory* names *dnames*:

$$
\begin{array}{|l}
\hline
\textit{NS} \\
NS0 \\
dnames : \mathbb{P}\, NAME \\
\hline
front(\!|\ dnames \cup \mathrm{dom}\, nstore\ |\!) \subseteq dnames \\
\hline
\end{array}
$$

$\mathbb{P}$ is the *powerset* constructor. The fat brackets $(\!|\,|\!)$ denote application of the function (*front* in this case) to a *set* of arguments to yield a *set* of results. That is,

$$front(\!|\ S\ |\!) = \{s : S \mid s \in \mathrm{dom}\, front \bullet front(s)\}$$

The *front* of a sequence is obtained by removing its last element; only the empty sequence ('root') has no *front*. The predicate states that the *front* of every (file or directory) name must itself be a directory name (i.e. every file or directory – except root – must appear in some directory). For example, if dom *nstore* included

/Carroll/Unix/paper
/dev/sanders
/Bernard/IEEE/Unixpaper
/Bernard/Mumble

(where syllables are preceded by /) then *dnames* would necessarily include

/
/Carroll
/dev
/Bernard
/Carroll/Unix
/Bernard/IEEE

Given a directory name *dir*?, the operation *lsNS* reveals its 'contents'.

$$
\begin{array}{|l}
\hline
\textit{lsNS} \\
\Xi NS \\
dir? : NAME \\
contents! : \mathbb{P}\, SYL \\
\hline
dir? \in dnames \\
contents! = last(\!|\ \{n : \mathrm{dom}\, nstore \mid n \neq \langle\rangle \land front\ n = dir?\}\ |\!) \\
\hline
\end{array}
$$

The *last* of a sequence is its final element.

4.3.11 Directories are files

An additional constraint on the Unix system is that directories are in fact stored as files; they can be read by users. That is,

$$dnames \subseteq \operatorname{dom} nstore$$

dirformat is a function that maps a *FILE* to the directory structure it represents:

$$\begin{array}{|l} dirformat : FILE \rightarrowtail\!\!\!\!\!\!\!\twoheadrightarrow (SYL \rightarrowtail\!\!\!\!\!\!\!\twoheadrightarrow FID) \\ RootFid : FID \end{array}$$

The mathematical definition of *dirformat would be* the definition of the format of a directory file – but such a definition need not be given here. *RootFid* is the *FID* of the root directory $\langle\ \rangle$. The content of each directory file is determined by the system in accordance with the following requirement:

$$\begin{array}{|l} \textit{dirstored} \\ \hline SS \\ NS \\ \hline \operatorname{ran} nstore \subseteq \operatorname{dom} fstore \\ nstore = (\lambda\, n : NAME \mid n \neq \langle\rangle \wedge n \in \operatorname{dom} nstore \bullet \\ \qquad\qquad (dirformat(fstore(nstore(front\ n))))(last\ n)) \\ \qquad\quad \cup \{\langle\ \rangle \mapsto RootFid\}. \\ \hline \end{array}$$

The constraint above may be paraphrased as follows:

> The association of names and file identifiers (*nstore*) is found by taking for any name ($\lambda\, n : NAME \ldots$) all of its syllables except the last (*front n*); finding the file identifier so referred to (*nstore* ...); finding the contents of that file (*fstore* ...); interpreting those contents as a directory (*dirformat* ...); and finally using the last syllable of the original name (*last n*) to obtain a file identifier from that directory – unless the original name is empty, in which case its file identifier is *RootFid*.

4.3.12 The complete filing system

The complete filing system is described by combining the descriptions of the three separate systems above: the storage systems *SS*, the channel system *CS*, and the name system *NS*.

$$\begin{array}{|l} \textit{FS} \\ \hline SS \\ CS \\ NS \\ usedfids : \mathbb{P}\, FID \\ \hline usedfids = \operatorname{ran} nstore \cup \{chan : \operatorname{ran} cstore \bullet chan.fid\} \\ usedfids \subseteq \operatorname{dom} fstore \\ \hline \end{array}$$

The auxiliary observation *usedfids* is introduced; it is the set of file identifiers in use at any time, either in the channel store or the name store. The predicate states that all file identifiers in use must refer to an existing file in the file store; members of $(\mathrm{dom}\, \mathit{fstore} \setminus \mathit{usedfids})$ are the *fids* of files which may be destroyed (since they are not referred to).

The filing system operations can be specified by combining the definitions of their effects on each separate subsystem. The *createFS* operation, for example, makes an empty file in the storage system, a new channel referring to it in the channel system, and associates a name with it in the naming system.

$$
\begin{array}{|l}
\hline
\mathit{createFS0} \\
\hline
\Delta \mathit{FS} \\
\mathit{createSS} \\
\mathit{openCS} \\
\mathit{createNS} \\
\hline
\mathit{name?} \in \mathrm{dom}\, \mathit{nstore} \Rightarrow \mathit{fid} = \mathit{nstore}(\mathit{name?}) \\
\mathit{name?} \notin \mathrm{dom}\, \mathit{nstore} \Rightarrow \mathit{fid} \notin \mathit{usedfids} \\
\hline
\end{array}
$$

If an *existing* name is created, the file it refers to is emptied – i.e. it is simply replaced with an empty file, and its previous contents are lost. If the name does not exist in the naming system, a new *fid* is chosen which is not currently in use.

The channel identifier of a channel referring to the new (or newly truncated) file is returned ($\mathit{cid!}$ is an observation of *openCS*).

The definition of *createFS* above is not sufficient. Remember that the name store is encoded in the file store as directory files. In the case where a new name is added to the name store, it also needs to be added to the (encoded) directory in the file store. We define the following schema, which updates the directory files in the file store without changing the non-directory files or the name store. It makes use of the schema *dirstored* on the final state to ensure the name store is correctly encoded into the file store.

$$
\begin{array}{|l}
\hline
\mathit{direncode} \\
\hline
\Delta \mathit{FS} \\
\Xi \mathit{CS} \\
\Xi \mathit{NS} \\
\mathit{dirstored}' \\
\hline
\exists\, \mathit{dfids} : \mathbb{P}\, \mathit{FID} \bullet \mathit{dfids} = \mathit{nstore}(\!|\, \mathit{dnames} \,|\!) \land \\
\qquad \mathit{dfids} \lhd \mathit{fstore}' = \mathit{dfids} \lhd \mathit{fstore} \\
\hline
\end{array}
$$

The only difference between the file store before encoding and the file store after is the contents of directory files. Before encoding they may not accurately represent the name store but afterwards they must.

The definition of *createFS* can now be completed. It is the schema composition ($\fatsemi$) of *createFS0* and *direncode*. The definition of schema composition is given in Section 4.3.14.

$$\mathit{createFS} \mathrel{\widehat{=}} \mathit{createFS0} \fatsemi \mathit{direncode}$$

open returns the channel identifier of an existing file.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{openFS}} \\
\hline
\Delta FS \\
\Xi SS \\
openCS \\
lookupNS \\
\hline
fid = fid' \\
\hline
\end{array}
$$

The fid' returned by *lookupNS* is equal to the *fid* supplied to *openCS* (and both fid' and *fid* are good candidates for hiding).

read and *write* do not change the name store.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{readFS}} \\
\hline
\Delta FS \\
readAS \\
\Xi NS \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{writeFS}} \\
\hline
\Delta FS \\
writeAS \\
\Xi NS \\
\hline
\end{array}
$$

close removes the association between a channel name and the channel it refers to.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{closeFS}} \\
\hline
\Delta FS \\
\Xi SS \\
closeCS \\
\Xi NS \\
\hline
\end{array}
$$

unlink removes a name from the naming system, but it does *not* destroy the associated file.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{unlinkFS0}} \\
\hline
\Delta FS \\
\Xi SS \\
\Xi CS \\
destroyNS \\
\hline
\end{array}
$$

As with *createFS*, this operation updates the name store. Hence the encoded version of the name store in the file store also needs to be updated.

$$unlinkFS \mathrel{\widehat=} unlinkFS0 \mathbin{⨾} direncode$$

Destroy removes a file from the filing system.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{destroyFS}} \\
\hline
\Delta FS \\
destroySS \\
\Xi CS \\
\Xi NS \\
\hline
\end{array}
$$

But can a file be destroyed while it is in use? The FS' invariant requires that

$$usedfids' \subseteq \mathrm{dom}\, fstore' \tag{4.1}$$

and from ΞCS and ΞNS it follows that

$$usedfids = usedfids' \tag{4.2}$$

and so, from (4.1) and (4.2),

$$usedfids \subseteq \mathrm{dom}\, fstore' \tag{4.3}$$

But

$$destroySS \Rightarrow fid \notin \mathrm{dom}\, fstore' \tag{4.4}$$

and (4.3) and (4.4) give

$$destroySS \Rightarrow fid \notin usedfids \tag{4.5}$$

That is, a file cannot be destroyed while it is in use.

4.3.13 Honesty of definitions

The constraint on the destroy operation

$$fid \notin usedfids$$

is not immediately obvious from its definition above. Because the constraint is implicit, the above definition could be said to be dishonest.

An honest definition is one for which the conditions of applicability are explicit. In general, a schema which describes an operation can be expanded to have the form

$$\begin{array}{|l}
\textit{operation} \\
\hline
STATE \\
STATE' \\
IN? \\
OUT! \\
\hline
inv(STATE) \\
pre(STATE, IN?) \\
trans(STATE, IN?, OUT!, STATE') \\
post(STATE', OUT!) \\
inv(STATE') \\
\hline
\end{array}$$

where $P(S)$ denotes a predicate in which the observations of S may occur free.

$STATE$, $STATE'$, $IN?$, and $OUT!$ are schemas with no predicates – they are just signatures.

inv is the state invariant, *pre* and *post* are the pre- and post-conditions respectively, and *trans* is the predicate expressing the relationship between the initial state, inputs, outputs, and final state. The conjunction of the five predicates forms the definition of the operation, but the definition is said to be *honest* only if

$$inv \land pre \Rightarrow (\exists\, OUT!;\ STATE' \bullet trans \land post \land inv)$$

If the invariant holds, and the input satisfies its precondition, then the operation should have at least one defined result. Thus, in an honest definition, applicability can be determined by considering the precondition alone (if all operations preserve the invariant). This is an honest definition of destroy:

$$
\begin{array}{|l}
\hline
\textit{destroyFS} \\
\Delta FS \\
\textit{destroySS} \\
\Xi CS \\
\Xi NS \\
\hline
\textit{fid} \notin \textit{usedfids} \\
\hline
\end{array}
$$

It is, however, *mathematically* equivalent to its original definition above.

For any schema describing an operation, a suitably honest precondition can be discovered by hiding the $OUT!$ and $STATE'$ observations, and simplifying the resulting predicate.

4.3.14 Observation renaming and schema composition

It may be necessary at times to rename the observations of a schema to avoid name clashes with other schemas. Writing

$$\textit{schema}[\textit{name}2/\textit{name}1]$$

denotes the result of systematically substituting $\textit{name}2$ for $\textit{name}1$ throughout $\textit{schema}$ (with suitable renaming of bound variables if necessary). For example:

$$\textit{createNS}[\textit{newname}?/\textit{name}?] =$$

$$
\begin{array}{|l}
\hline
\Delta NS \\
\textit{newname}? : NAME \\
\textit{fid} : FID \\
\hline
\textit{nstore}' = \textit{nstore} \oplus \{\textit{newname}? \mapsto \textit{fid}\} \\
\hline
\end{array}
$$

and

$$\textit{lookupNS}[\textit{oldname}?/\textit{name}?] =$$

$$
\begin{array}{|l}
\hline
\Delta NS \\
\textit{oldname}? : NAME \\
\textit{fid}' : FID \\
\hline
\textit{oldname}? \in \operatorname{dom} \textit{nstore} \\
\textit{fid}' = \textit{nstore}(\textit{oldname}?) \\
\textit{nstore}' = \textit{nstore} \\
\hline
\end{array}
$$

The *composition* of two schemas, written

$$schema1 \semi schema2$$

is intended to capture the effect of '*schema*1 then *schema*2'. It is formed by

1. Determining all of the dashed observations of *schema*1 that correspond with undashed observations of *schema*2 ($name'$ corresponds with $name$).
2. Renaming each corresponding pair to a single new name

$$schema1[name''/name']$$
$$schema2[name''/name]$$

3. Combining the schemas, and hiding the new observations

$$\begin{aligned} &schema1 \semi schema2 \mathrel{\widehat=} \\ &\quad (schema1[name''/name'] \land \\ &\qquad schema2[name''/name]) \setminus (name''). \end{aligned}$$

This operation allows schemas to be combined in a way suggestive of forward functional composition: the final state of *schema*1 becomes the initial state of *schema*2. For example:

$$\begin{aligned} &linkNS \mathrel{\widehat=} lookupNS[oldname?/name?] \semi \\ &\qquad createNS[newname?/name?] \end{aligned}$$

gives in full:

$$\begin{array}{|l} \hline \textit{linkNS} \\ \Delta NS \\ oldname?, newname? : NAME \\ \hline oldname? \in \operatorname{dom} nstore \\ nstore' = nstore \oplus \{newname? \mapsto nstore(oldname?)\} \\ \hline \end{array}$$

The hidden observations are *nstore* and *fid*. *linkNS* makes the filename *newname*? refer to the same file as does *oldname*?.

A similar construction defines *moveNS*:

$$moveNS \mathrel{\widehat=} linkNS \semi destroyNS[oldname?/name?]$$

That is,

$$\begin{array}{|l} \hline \textit{moveNS} \\ \Delta NS \\ oldname?, newname? : NAME \\ \hline oldname? \in \operatorname{dom} nstore \\ nstore' = (\{oldname?\} \ndres nstore) \oplus \{newname? \mapsto nstore(oldname?)\} \\ \hline \end{array}$$

moveNS renames a file from *oldname*? to *newname*?. It is important that the two occurrences of *oldname*? – in *linkNS* and *destroyNS*[*oldname*?/*name*?] – are merged, and so only one file is referred to. However, $oldname?$ appears only once in the signature of $moveNS$.

Combining the definitions of $linkNS$ and $moveNS$ above, with ΞSS and ΞCS, gives their definitions in the complete file system FS.

$$\begin{aligned} linkFS &\mathrel{\widehat=} (\Delta FS \wedge \Xi SS \wedge \Xi CS \wedge linkNS) \fatsemi direncode \\ moveFS &\mathrel{\widehat=} (\Delta FS \wedge \Xi SS \wedge \Xi CS \wedge moveNS) \fatsemi direncode \end{aligned}$$

Because both these operations update the name store, we need to update the encoded version of the name store in the file store.

4.3.15 Definition of error conditions

The definitions given so far describe only *successful* operations. For example, the schema

$$\begin{array}{|l} \textit{lookupNS} \\ \hline \Xi NS \\ name? : NAME \\ fid' : FID \\ \hline name? \in \operatorname{dom} nstore \\ fid' = nstore(name?) \\ \hline \end{array}$$

gives no indication of the result of looking up a name that is *not* in the name store. In fact, the definition explicitly states that the name must be there

$$name? \in \operatorname{dom} nstore.$$

It is to that extent unrealistic.

To describe unsuccessful as well as successful operations, a schema is introduced below which includes an *error report* observation. The following error reports are used:

$$REPORT ::= Ok \mid NoSuchCid \mid NoSuchName \mid NoFreeCids$$

$$\begin{array}{|l} \Delta FS \\ \hline FS \\ FS' \\ report! : REPORT \\ \hline report! \neq Ok \Rightarrow (\theta FS' = \theta FS) \\ \hline \end{array}$$

The predicate states that in the event of an unsuccessful report

$$report! \neq Ok$$

the system's state is unaltered ($\theta FS' = \theta FS$). Successful operations are described by the schema below:

$$\begin{array}{|l} \textit{success} \\ \hline \Delta FS \\ \hline report! = Ok \\ \hline \end{array}$$

The following schemas define typical failures:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CidErr}} \\
\Delta FS \\
cid? : CID \\
\hline
cid? \notin \operatorname{dom} cstore \\
report! = NoSuchCid \\
\hline
\end{array}$$

CidErr describes an attempt to use a non-existent channel identifier. Two other common errors are

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{NameErr}} \\
\Delta FS \\
name? : NAME \\
\hline
name? \notin \operatorname{dom} nstore \\
report! = NoSuchName \\
\hline
\end{array}$$

and

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ChanErr}} \\
\Delta FS \\
\hline
\operatorname{dom} cstore = CID \\
report! = NoFreeCids \\
\hline
\end{array}$$

NameErr describes an attempt to use a non-existent file name; *ChanErr* describes an unsuccessful attempt to obtain a new channel identifier.

These error descriptions should be associated with the operations that can give rise to them; this is accomplished by schema *disjunction*:

$$schema1 \lor schema2$$

This is the schema formed by merging the two schemas' signatures (as for conjunction $\land$) and forming the disjunction of their predicate parts (where, in contrast, $\land$ forms their conjunction).

Thus the schemas *read* and *open*, for example, can be redefined to include the error cases:

$$read \mathrel{\widehat{=}} (readFS \land success) \lor CidErr$$
$$open \mathrel{\widehat{=}} (openFS \land success) \lor NameErr \lor ChanErr$$

The other operations may be similarly treated once their error conditions have been defined.

Figures 4.1 and 4.2 give the expansions of *read* and *open*, respectively.

4.4 Summary

The schema approach to the incremental presentation of large system specifications has been illustrated by using it to describe the Unix filestore. This technique has

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad read \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
FS \\
FS' \\
cid? : CID \\
length? : \mathbb{N} \\
data! : \mathrm{seq}\, BYTE \\
report! : REPORT \\
CHAN \\
file : FILE \\
\hline
(report! = Ok \land \\
\qquad cid? \in \mathrm{dom}\, cstore \land \\
\qquad \theta CHAN = cstore(cid?) \land \\
\qquad file = fstore(fid) \land \\
\qquad fstore' = fstore \land \\
\qquad (\exists\, CHAN' \bullet posn' = posn + \#data! \land \\
\qquad\qquad fid' = fid \land \\
\qquad\qquad cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\}) \land \\
\qquad nstore' = nstore \land \\
\qquad data! = (1 \mathinner{\ldotp\ldotp} length?) \lhd (file \ \mathrm{after}\ posn)) \\
\lor\ (report! = NoSuchCid \land \\
\qquad cid? \notin \mathrm{dom}\, cstore \land \\
\qquad \theta FS' = \theta FS) \\
\hline
\end{array}
$$

Figure 4.1: Expansion of *read*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{open}} \\
\hline
FS \\
FS' \\
name? : NAME \\
cid! : CID \\
report! : REPORT \\
fid, fid' : FID \\
\hline
(report! = Ok \land \\
\quad name? \in \operatorname{dom} nstore \land \\
\quad fid = fid' = nstore(name?) \land \\
\quad fstore' = fstore \land \\
\quad (\exists\, CHAN' \bullet posn' = 0 \land \\
\quad\quad fid' = fid \land \\
\quad\quad cstore' = cstore \oplus \{cid! \mapsto \theta CHAN'\}) \land \\
\quad nstore' = nstore \land \\
\quad cid! \notin \operatorname{dom} cstore) \\
\lor\ (report! = NoSuchName \land \\
\quad name? \notin \operatorname{dom} nstore \land \\
\quad \theta FS' = \theta FS) \\
\lor\ (report! = NoFreeCids \land \\
\quad \operatorname{dom} cstore = CID \land \\
\quad \theta FS' = \theta FS) \\
\hline
\end{array}
$$

Figure 4.2: Expansion of *open*

been used elsewhere to present, and reason about, specifications of other large-scale systems [40, 41, 60, 62]. It has also proved useful in presenting the behaviour of systems from a *variety* of points of view, drawing these together by showing how they are related from an 'Olympian' point of view.

However, because of the generality of the underlying theory (set theory), and in particular because of the unrestricted nature of the predicates which can be written to characterise operations, there is no *a priori* guarantee that a system specified in this style is implementable, nor is there any 'automatic' way of checking even its internal consistency. The best that can be done is to demonstrate a constructive model at a suitably high level of abstraction. Fortunately, the provision of such a model is usually the first step to be taken in the development of an implementation.

This specification technique is not yet a *development method;* it is simply a step on the way to one. In particular, the usual criteria for deciding on correctness of representations and of algorithms have yet to be adapted to this style of presentation.

Once suitable mathematical types have been discovered for the *observations* to be made of a system (i.e. once suitable mathematical theories have been found and decided upon), the *narrative* part of the top-level views of a system is relatively easy to formulate.

Acknowledgements The use of set theory to specify the behaviour of computer systems was first explained to us by Jean-Raymond Abrial. This specification has been developed from the original attempt by Richard Miller. We have benefited from collaboration with many of our colleagues, especially Ib Sørensen, Steve Schumann, Tony Hoare, Ian Hayes, Roger Gimson, and Tim Clement. The continuing financial support of the UK Science and Engineering Research Council is gratefully appreciated.

4.5 Appendix: differences from Unix

4.5.1 File size

There is an upper bound on the size of files; if a file could contain no more than *FileSizeLimit* bytes

$$FileSizeLimit : \mathbb{N}_1$$

then *FILE* would be defined

$$FILE == \{f : \mathrm{seq}\, BYTE \mid \# f \leq FileSizeLimit\}$$

4.5.2 Directory size

There is an upper bound on the number of files in the storage system (i.e. the number of 'inodes' is limited):

$$FileNumberLimit : \mathbb{N}_1$$

$$\begin{array}{|l}
\hline
SS \\
\hline
fstore : FID \nrightarrow FILE \\
\hline
\# fstore \leq FileNumberLimit \\
\hline
\end{array}$$

4.5.3 Storage medium capacity

The storage medium used to implement the filing system has finite capacity:

$$\mid \mathit{DeviceCapacity} : \mathbb{N}_1$$

We assume *minbytes*

$$\mid \mathit{minbytes} : \mathit{FILE} \rightarrow \mathbb{N}$$

maps a file into the minimum number of bytes required to represent it in the storage system.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SS}} \\
\hline
\mathit{fstore} : \mathit{FID} \nrightarrow \mathit{FILE} \\
\hline
\#\mathit{fstore} \leq \mathit{FileNumberLimit} \\
\mathit{DeviceCapacity} \geq \sum [\![\mathit{fid} : \mathrm{dom}\, \mathit{fstore} \bullet \mathit{minbytes}(\mathit{fstore}\ \mathit{fid})]\!] \\
\hline
\end{array}$$

Because in the storage system it is possible to represent a file in more than one way (small, large, huge – also, totally zero blocks may or may not be allocated), all that can be said about the system's capacity is that it must be at least as large as the minimum required to represent the files within it. Similarly, all that can be said of the device-full condition is that it *cannot* occur while the capacity is sufficient for the *maximum* required. We assume *maxbytes*

$$\mid \mathit{maxbytes} : \mathit{FILE} \rightarrow \mathbb{N}$$

maps a file into the maximum number of bytes required to represent it in the storage system. The condition

$$\sum [\![\mathit{fid} : \mathrm{dom}\, \mathit{fstore}'' \bullet \mathit{maxbytes}(\mathit{fstore}''\ \mathit{fid})]\!] > \mathit{DeviceCapacity}$$

is *necessary* for a device full error (where fstore'' is the storage system which would have resulted from the attempted operation).

4.5.4 Seek

seek as defined in Unix has several options, which automatically calculate the desired new offset depending, for example, on the file's current length. These may be described separately.

$$
\begin{array}{|l}
\hline
\textit{seekoffset} \\
SS \\
CHAN \\
n? : \mathbb{N} \\
p? : 0 \,..\, 5 \\
\textit{offset}?, \\
\textit{size} : \mathbb{N} \\
\hline
\textit{size} = \#(\textit{fstore}(\textit{fid})) \\
p? = 0 \Rightarrow \textit{offset}? = n? \\
p? = 1 \Rightarrow \textit{offset}? = \textit{posn} + n? \\
p? = 2 \Rightarrow \textit{offset}? = \textit{size} + n? \\
p? = 3 \Rightarrow \textit{offset}? = 512 * n? \\
p? = 4 \Rightarrow \textit{offset}? = \textit{posn} + 512 * n? \\
p? = 5 \Rightarrow \textit{offset}? = \textit{size} + 512 * n? \\
\hline
\end{array}
$$

offset? and *size* are now auxiliary components.

The above schema could be combined with the schema for *seek* to give the full definition of the seek system call.

4.5.5 Representation of numbers

The new position of the file is in fact limited by the ability of the computer to represent numbers.

In this and other cases this limitation could be expressed, for example, as:

$$n24bit == 0 \,..\, 2^{24} - 1$$

Such sets would then be used, where appropriate, instead of $\mathbb{N}$:

$$
\begin{array}{|l}
\hline
CHAN \\
\textit{fid} : FID \\
\textit{posn} : n24bit \\
\hline
\end{array}
$$