9 10 10.95 12

A Specification for the Synergy File System

William R. Bevier
Richard Cohen
Jeff Turner [1]

Technical Report 120 September, 1995

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: bevier@cli.com, cohen@cli.com, sjt@tycho.ncsc.mil

[1] National Security Agency, R23

Contents

1 Introduction

This document contains a Z specification [Spivey, 1989] for a subset of the user interface functions of the Synergy file system. The Synergy file system is intended to be compatible with the Unix file system, but include additional constraints on behavior related to security. Morgan and Sufrin have previously specified the Unix file system interface [Morgan and Sufrin, 1987]. We have re-specified this interface to include features they did not treat, for example, discretionary access control. Our specification is influenced by the descriptions of the Unix file system given by *The Design of the Unix Operating System* [Bach, 1986] and *The Design and Implementation of the 4.3BSD UNIX Operation System*[Leffler et al., 1989]. Additional details have been gleaned from the *UNIX User's Reference Manual* for 4.3BSD [CSRG, 1986] and Steven's treatise on Unix programming [Stevens, 1992].

The work described by this paper was performed as a subproject of the Synergy project, whose primary goal is to develop policy-flexible security architectures that meet the needs of both the DoD (separation of data with high assurance) and the commercial world (flexibility, larger market than DoD). This work will serve as a foundation for several future projects. First, we will continue to articulate this specification in order to fully capture the design and interface of a policy-independent file system interface. Ultimately, this document should server as a programmer's manual for the file system. Second, we will use it as the highest level specification in a proof that the decomposition of the interface into a number of independent servers is valid.

In building a model of a system one must make choices regarding level of abstraction. These choices reveal properties of interest and ignore others. We are interested in mandatory and discretionary access controls on files. We are less interested in covert channels that may be created by these mechanisms. We therefore ignore limitations on virtually all resources. Such limitations can be introduced gradually to allow analysis of the channels they produce.

We have left error reporting and return codes out of our model. The model does specify the preconditions necessary for successful operations within the file system. We simply do not describe the effect of an operation if its preconditions are not met.

Section 2 of this report contains definitions of some primitive Z functions that are used elsewhere. Subsequent sections incrementally specify aspects of the file system interface. The following table summarizes the concepts that are introduced.

State	Concepts Introduced	Operations Introduced
Basic File Space	files, file identifiers	create, destroy, read, write
File Space	directories, pathnames	link, unlink
File Table	access modes, file position	open, close
Processes	processes, file descriptors	fork process, duplicate fd
Garbage Collection	reclaiming inaccessible files	
Basic Access System		
DAC	user, group, file permissions	
Unix FS Interface	current process, working directory DAC checks for operations	
MAC	security context	
Synergy FS Interface	MAC checks for operations	get/set context

2 Preliminaries

We presume the reader is familiar with the Z notation (e.g., has access to the *Z Reference Manual* [Spivey, 1989]).

Naming Conventions

In general the words "create" and "destroy", "link" and "unlink" are used to describe file operations. The words "add" and "delete" are used to describe operations on other data structures/state components. One exception is the use of "create" and "destroy" to name the operations that create and destroy file and process attributes in the DAC and MAC layers.

Some Z preliminaries

In the Z notation, a sequence of length $\mathbf{n}$ is a mapping from natural numbers between 1 and $\mathbf{n}$ to the individual elements of the sequence. That is, sequence indexing is 1-origin.

The function **Nthtail** returns the subsequence of a sequence beginning at a given offset. Essentially, it skips $\mathbf{n}$ things, and gives a new sequence starting there. Note that for all sequences $\mathbf{s}$, $\mathbf{nthtail}(0, \mathbf{s}) = \mathbf{s}$. That is, an offset of 0 skips nothing at the beginning of a sequence. This is equivalent to Morgan and Sufrin's function **after**.

$$
\begin{array}{|l}
[\mathbf{X}] \\
\hline
\mathbf{nthtail} : \ \times \operatorname{seq} \mathbf{X} \to \operatorname{seq} \mathbf{X} \\
\hline
\forall \mathbf{n} : ; \ \ \mathbf{s} : \operatorname{seq} \mathbf{X} \bullet \\
\quad \operatorname{dom}(\mathbf{nthtail}(\mathbf{n}, \mathbf{s})) = (1 \mathinner{..} \#\mathbf{s} - \mathbf{n}) \ \ \land \\
\quad (\forall \mathbf{i} : \ \bullet \\
\qquad (\mathbf{i} + \mathbf{n}) \in \operatorname{dom} \mathbf{s} \Rightarrow (\mathbf{nthtail}(\mathbf{n}, \mathbf{s}))(\mathbf{i}) = \mathbf{s}(\mathbf{i} + \mathbf{n})) \\
\hline
\end{array}
$$

Shift adjusts the domain of a sequence forward by some offset. Morgan and Sufrin define the same function, but make it an infix operator.

$$
\begin{array}{|l}
[\mathbf{X}] \\
\hline
\mathbf{shift} : \ \times \operatorname{seq} \mathbf{X} \to (\ \ \mathbf{X}) \\
\hline
\forall \mathbf{n} : ; \ \ \mathbf{s} : \operatorname{seq} \mathbf{X} \bullet \\
\quad \operatorname{dom}(\mathbf{shift}(\mathbf{n}, \mathbf{s})) = \{\mathbf{i} : \operatorname{dom} \mathbf{s} \bullet \mathbf{i} + \mathbf{n}\} \ \ \land \\
\quad (\forall \mathbf{i} : \operatorname{dom}(\mathbf{shift}(\mathbf{n}, \mathbf{s})) \bullet \\
\qquad (\mathbf{shift}(\mathbf{n}, \mathbf{s}))(\mathbf{i}) = \mathbf{s}(\mathbf{i} - \mathbf{n})) \\
\hline
\end{array}
$$

Recall that the sequence $\langle \mathbf{a}, \mathbf{b} \rangle$ denotes the mapping $\{\ 1 \mapsto \mathbf{a},\ 2 \mapsto \mathbf{b}\ \}$. The expression $\mathbf{shift}(3,\ \langle \mathbf{a}, \mathbf{b} \rangle)$ denotes the mapping $\{(3+1) \mapsto \mathbf{a},\ (3+2) \mapsto \mathbf{b}\}$, which is $\{4 \mapsto \mathbf{a},\ 5 \mapsto \mathbf{b}\}$).

Remarks on Z style

We like schema composition: the pipe () and semi-colon () operators. We think of compositions as data-flow diagrams. Pipes connect input and output variables; semi-colon connects initial and final states.

What does this view mean about our style of Z usage? We tend to define intermediate schemas for one of four uses:

- system states,
- state transitions, for use with semi-colon composition,
- data translations, for use with pipes, or
- predicates, for use with logical connectives.

Each of these uses has a different characteristic style. State transitions typically include Ξ**State** or Δ**State** for some schema **State**. Data translations typically include both input and output variables, with some axioms defining a relation between them. A predicate typically has input variables, but no output variables. It includes an axiom part (i.e., "the part below the line") that describes some required properties of the input variables and/or the system state.

Of course, we also define schemas that are merely intended to encapsulate state or define some functions or sets that will be included in later schemas.

Things Left Out of the Model

Our intent is to model significant aspects of the user-program interface to the Synergy file system. Since the Synergy file system interface includes the Unix file system interface as a subset, our model includes aspects of UFS. Our model is not intended to be complete. It neither models all of the user-level operations of the Synergy (or the Unix) file system, nor does it model all the details of every operation included in the model.

Here is a list of some prominent aspects of UFS that we have not included in the SFS model. Some have been left out merely because of time constraints. Some were aspects judged not necessary to produce a useful model. Some were considered unimportant "implementation details" of UFS that show through at the interface level. We have not modeled these aspects, but our model should admit consistent extensions that do include some or all of these aspects.

1. error reporting
2. any resource limits (e.g., size of pathnames, *etc.*)
3. chdir, chroot (at least not yet)
4. chown, chgroup (at least not yet)
5. chmod (at least not yet)
6. mkdir, rmdir
7. mount points to multiple file systems
8. read-only file systems
9. real_uid versus effective_uid
10. real_gid versus effective_gid
11. execution of files (and, hence, the set_uid or set_gid permissions)
12. the directory entries "." and ".."
13. super-user privileges or restrictions.

14. special files (not even their presence in the file system)
15. soft links (also called symbolic links)
16. file or inode creation/access/modification times
17. blocking or non-blocking accesses
18. some open modes (e.g., excl)
19. locking
20. the 4.2 BSD directory manipulation operations: opendir, readdir, rewinddir, closedir, telldir, seekdir.
21. the stat system call
22. possible interaction between concurrent reading and writing of a file. (Currently a file is truncated to zero length when it is opened for writing. See the definition of **OpenFT**, page 15.)

3 The Basic File Space

A file is a sequence of bytes.

$[\mathbf{BYTE}]$

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

The function **padfile** constructs a file containing a sequence of pad bytes to a given length. Usually in Unix, the pad byte is a representation of the number 0.

$$\begin{array}{|l} \mathbf{pad} : \mathbf{BYTE} \\ \mathbf{padfile} : \ \rightarrow \mathbf{FILE} \\ \hline \forall \mathbf{n} : \ \bullet \ \mathbf{padfile}(\mathbf{n}) = (\lambda \mathbf{k} : 1 \,..\, \mathbf{n} \bullet \mathbf{pad}) \end{array}$$

A file identifier is the internal name of a file. The Unix implementation of a file identifier is an inode number.

$[\mathbf{FID}]$

The following schema introduces the association of a file identifier with the contents of a file. We specify four transitions on a basic file space: create, destroy, read and write.

$$\begin{array}{|l} \textbf{BasicFileSpace} \\ \hline \mathbf{fcontents} : \mathbf{FID} \ \ \mathbf{FILE} \\ \hline \end{array}$$

NewFID produces a file identifier not currently in use (i.e., a "new file identifier") as its output. Its output will be piped into other schemas that take a file identifier as input.

NewFid

BasicFileSpace
fid! : **FID**

---

fid! $\notin$ dom **fcontents**

FileLength takes a file identifier as input, and produces an output whose value is the length of the file's contents. The output is named **offset**.

FileLength

BasicFileSpace
fid? : **FID**
offset! :

---

fid? $\in$ dom **fcontents**

offset! = #(**fcontents**(**fid**?))

Create

File creation is a state transition on a **BasicFileSpace**. It takes a currently-unused file identifier as input, and causes it to be bound to an empty file in the file space.

CreateBFS

Δ**BasicFileSpace**
fid? : **FID**

---

fid? $\notin$ dom **fcontents**

fcontents$'$ = **fcontents** $\oplus$ {**fid**? $\mapsto$ $\langle\rangle$}

Destroy

When a file is destroyed, the association of a file identifier with the file's contents is removed from the file space. [1]

DestroyBFS

Δ**BasicFileSpace**
fid? : **FID**

---

fid? $\in$ dom **fcontents**

fcontents$'$ = {**fid**?} $\lhd$ **fcontents**

[1] In a typical implementation the disk blocks belonging to the file remain on disk. So removing the association of **fid** $\mapsto$ **file** reflects that the file is no longer accessible via the file system. Our model does not include the underlying storage medium (e.g., a disk system).

Read

The input values of a read operation are a file identifier, an offset where reading begins, and a length of data to be read. The output value is the sequence of bytes that occurs at the given offset in the file. The length of this sequence is the requested length or the remaining length to the end of the file, whichever is smaller. Thus, the length of the resulting sequence depends on the amount of data in the file.

ReadBFS

Ξ**BasicFileSpace**
fid? : **FID**
offset?, **length**? :
data! : **FILE**

---

fid? $\in$ dom **fcontents**

data! = (1 . . **length**?) **nthtail**(**offset**?, **fcontents**(**fid**?))

Write

The inputs to a write operation are a file identifier, an offset at which writing begins, and a sequence of bytes to be written. The new sequence of bytes replaces existing bytes in the designated range. The file length is increased if necessary, and if the offset is longer than the initial file length pad characters are inserted. [2]

WriteBFS

Δ**BasicFileSpace**
fid? : **FID**
offset? :
data? : **FILE**

---

fid? $\in$ dom **fcontents**

fcontents$'$ = **fcontents**$\oplus$
$\{$**fid**? $\mapsto$ **padfile**(**offset**?) $\oplus$ **fcontents**(**fid**?) $\oplus$ **shift**(**offset**?, **data**?)$\}$

4 The File Space

We introduce the notions of pathname and directory. A pathname is a sequence of syllables. A directory is a mapping from syllables to file identifiers.

[**SYLLABLE**]

PATHNAME == seq **SYLLABLE**

DIRECTORY == **SYLLABLE** $\pfun$ **FID**

[2] These pad characters correspond to the "holes" that Unix permits within files. These holes are read as pad characters (normally 0). Our model does not explicitly include holes, but does permit writing passed the existing file length, padding the file to the required offset before writing. Thus **read** will see the pad characters in the "hole", as in the Unix implementation.

We hypothesize the existence of a partial function **ParseDir** which creates an object of type **DIRECTORY** from a file (i.e., makes a file understandable as a directory). The total function **UnParseDir** performs a mapping from directories back to files (in order to store directories as files). This mechanism allows us to represent directories as files in system state, but use a convenient representation when performing "directory operations." **UnParseDir** is a total function because every directory object (i.e., mapping from name to file identifier) can be represented as a file (i.e., a sequence of bytes).

$$\begin{array}{|l} \mathbf{ParseDir} : \mathbf{FILE} \;\; \mathbf{DIRECTORY} \\ \mathbf{UnParseDir} : \mathbf{DIRECTORY} \to \mathbf{FILE} \\ \hline \operatorname{ran} \mathbf{UnParseDir} \subseteq \operatorname{dom} \mathbf{ParseDir} \\ \forall\, \mathbf{d} : \mathbf{DIRECTORY};\;\; \mathbf{s} : \mathbf{SYLLABLE} \bullet \\ \quad \operatorname{dom}(\mathbf{ParseDir}(\mathbf{UnParseDir}(\mathbf{d}))) = \operatorname{dom} \mathbf{d} \;\; \wedge \\ \quad (\mathbf{ParseDir}(\mathbf{UnParseDir}(\mathbf{d})))(\mathbf{s}) = \mathbf{d}(\mathbf{s}) \\ \operatorname{dom}(\mathbf{ParseDir}(\langle\rangle)) = \emptyset \end{array}$$

A file may be one of two types, **regular** or **directory**.[3] We apply the function **ParseDir** only to files of type **directory**.

$$\mathbf{FTYPE} ::= \mathbf{regular} \mid \mathbf{directory}$$

As a convenient abstraction, we define the partial function **DirContents** to map a **fid** identifying a directory file to the corresponding **DIRECTORY** value.

$$\begin{array}{|l} \mathbf{BasicFileSpace} \\ \mathbf{DirContents} : \mathbf{FID} \;\; \mathbf{DIRECTORY} \\ \hline \forall\, \mathbf{fid} : \mathbf{FID} \mid \mathbf{fid} \in \operatorname{dom} \mathbf{fcontents} \\ \qquad\qquad \wedge\, \mathbf{fcontents}(\mathbf{fid}) \in \operatorname{dom} \mathbf{ParseDir} \bullet \\ \quad \mathbf{DirContents}(\mathbf{fid}) = \mathbf{ParseDir}(\mathbf{fcontents}(\mathbf{fid})) \end{array}$$

The schema **FileSpace** introduces directories into the basic file space. Each file identifier has a file type. Files of type **directory** are interpreted as directories with the function **ParseDir**. The range of each directory file contains file identifiers occurring in the file space. There is a root file identifier that points to the top of the file tree.

The requirements on a file space are that:

- each file has a file type (i.e., **regular** or **directory**),
- the root file identifier names a directory file in the file space,
- each directory file can be interpreted as a directory structure,
- each directory provides names only for existing files,

[3] We do not include the Unix concept of special files in our model.

- a directory that is pointed to by no other directory must either be empty or be the root directory.[4]

This last requirement restricts the graph of directories to be a tree structures. We have not included specifications for the self and parent directory entries within a directory (i.e., "." and ".." in the Unix file system). [5]

FileSpace

$$
\begin{array}{l}
\textbf{BasicFileSpace} \\
\textbf{ftype} : \textbf{FID} \;\; \textbf{FTYPE} \\
\textbf{Rootfid} : \textbf{FID} \\
\hline
\mathrm{dom}\, \textbf{ftype} = \mathrm{dom}\, \textbf{fcontents} \\
\textbf{Rootfid} \in \mathrm{dom}\, \textbf{fcontents} \land \textbf{ftype}(\textbf{Rootfid}) = \textbf{directory} \\
\forall\, \textbf{fid} : \mathrm{dom}\, \textbf{fcontents} \bullet \\
\quad \textbf{ftype}(\textbf{fid}) = \textbf{directory} \Rightarrow \\
\qquad \textbf{fcontents}(\textbf{fid}) \in \mathrm{dom}\, \textbf{ParseDir} \\
\forall\, \textbf{fid} : \mathrm{dom}\, \textbf{fcontents} \bullet \\
\quad \textbf{ftype}(\textbf{fid}) = \textbf{directory} \Rightarrow \\
\qquad \mathrm{ran}(\textbf{DirContents}(\textbf{fid})) \subseteq \mathrm{dom}\, \textbf{fcontents} \\
\qquad \land\, (\textbf{parents} == \{\textbf{parent\_fid} : \mathrm{dom}\, \textbf{fcontents};\;\; \textbf{name} : \textbf{SYLLABLE} \mid \\
\qquad\quad \textbf{ftype}(\textbf{parent\_fid}) = \textbf{directory} \\
\qquad\quad \land\, \textbf{DirContents}(\textbf{parent\_fid})(\textbf{name}) = \textbf{fid} \bullet \\
\qquad\qquad \textbf{parent\_fid}\} \bullet \\
\qquad (\#\textbf{parents} = 0) \\
\qquad \Leftrightarrow (\textbf{fid} = \textbf{Rootfid} \lor \textbf{DirContents}(\textbf{fid}) = \emptyset))
\end{array}
$$

Remark on the Z

Note the property dom **ftype** = dom **fcontents**. *This is asserted wherever the* **FileSpace** *schema is inherited. Note further that* **ΔFileSpace** *asserts this property for* both *the* initial *and the* final *state variables. This means that we must be careful that any Create and Destroy operations that include* **ΔFileSpace** *must update both* **ftype** *and* **fcontents**.

We state that empty directories and regular files can be disconnected from the file space. Both regular files and directories may have multiple links (i.e., #**parents** > 1 in the definition of **FileSpace**).[6]

A file is named by a pathname and a starting directory. The pathname identifies the file relative to the starting directory.

The set **legal_pathnames** in a **FileSpace** contains the pair (**startfid**, **pn**) if and only if **pn** is a legal pathname relative to **startfid** in the file space.

[4] Unix file systems generally depend on a stronger version of this requirement: that a directory file is pointed to by at most one directory entry. Some Unix file system implementations permit this invariant to be violated (e.g., by a privileged user). However, they may not function properly when it is violated.

[5] The special directory entry ".." semi-violates the requirement that the directory structure be tree-structured.

[6] Most Unix systems only permit privileged processes to make new links to existing directories. It is not clear that it is every desirable to have multiple links to directories. We have tried merely to model what exists.

A pathname $(\mathbf{syl}_1, \mathbf{syl}_2, \ldots, \mathbf{syl_k})$ starting from a particular directory is legal in a file space either if it is the empty sequence or if the front of the pathname (i.e., excluding the last syllable) is a legal pathname that names a directory and that directory maps the last syllable to a file identifier. When pathname is legal, the function **pathname_fid** is the file identifier that pathname identifies.

The set **directory_fids** is the set of **fids** in the file space that identify directory files. (Specifically, it is the set of **fid** that the function **ftype** maps to **directory**.) This will be convenient, so that we can quantify over the directories in a file space.

LegalPathnames

$$\begin{array}{l}
\mathbf{FileSpace} \\
\mathbf{legal_pathnames} : (\mathbf{FID} \times \mathbf{PATHNAME}) \\
\mathbf{pathname_fid} : \mathbf{FID} \times \mathbf{PATHNAME} \;\; \mathbf{FID} \\
\mathbf{directory_fids} : \; \mathbf{FID} \\
\hline
\mathbf{directory_fids} = \mathbf{ftype}^{\sim}\{\mathbf{directory}\} \\
\mathrm{dom}\, \mathbf{pathname_fid} \subseteq \mathbf{directory_fids} \times \mathbf{PATHNAME} \\
\forall\, \mathbf{fid} : \mathbf{FID} \mid \mathbf{ftype}(\mathbf{fid}) = \mathbf{directory} \bullet \\
\quad \mathbf{pathname_fid}(\mathbf{fid}, \langle\rangle) = \mathbf{fid} \\
\forall\, \mathbf{fid} : \mathbf{FID};\;\; \mathbf{pn} : \mathbf{PATHNAME};\;\; \mathbf{name} : \mathbf{SYLLABLE} \mid \\
\qquad ((\mathbf{fid}, \mathbf{pn}) \in \mathrm{dom}\, \mathbf{pathname_fid} \\
\qquad \wedge\, \mathbf{ftype}(\mathbf{pathname_fid}(\mathbf{fid}, \mathbf{pn})) = \mathbf{directory} \\
\qquad \wedge\, \mathbf{name} \in \mathrm{dom}(\mathbf{DirContents}(\mathbf{pathname_fid}(\mathbf{fid}, \mathbf{pn})))) \bullet \\
\quad \mathbf{pathname_fid}(\mathbf{fid}, \mathbf{pn}\ \langle\mathbf{name}\rangle) = \mathbf{DirContents}(\mathbf{pathname_fid}(\mathbf{fid}, \mathbf{pn}))(\mathbf{name}) \\
\mathbf{legal_pathnames} = \mathrm{dom}\, \mathbf{pathname_fid}
\end{array}$$

Remark on the Z

The expression $\mathbf{ftype}^{\sim}\{\mathbf{directory}\}$ *denotes the inverse image of the singleton set containing the FTYPE value* **directory**. *That is, it represents the set* $\{\ \mathbf{fid} : \mathbf{FID} \mid \mathbf{ftype}(\mathbf{fid}) \in \{\mathbf{directory}\}\ \}$

The schema **FileSpace** includes the property that

$$\begin{array}{l}
\forall\, \mathbf{fid} : \mathrm{dom}\, \mathbf{fcontents} \bullet \\
\quad \mathbf{ftype}(\mathbf{fid}) = \mathbf{directory} \Rightarrow \\
\qquad \mathrm{ran}(\mathbf{DirContents}(\mathbf{fid})) \subseteq \mathrm{dom}\, \mathbf{fcontents}
\end{array}$$

From this we can deduce that

$$\mathrm{ran}\, \mathbf{pathname\_fid} \subseteq \mathrm{dom}\, \mathbf{fcontents},$$

because the only **fids** defined to be in the range of **pathname_fid** are those that are in the range of a parsed directory.

LegalPathname

$$\begin{array}{l}
\mathbf{LegalPathnames} \\
\mathbf{startfid?} : \mathbf{FID} \\
\mathbf{pn?} : \mathbf{PATHNAME} \\
\hline
(\mathbf{startfid?}, \mathbf{pn?}) \in \mathbf{legal_pathnames}
\end{array}$$

The schema **LegalPathname** *is designed as a predicate schema.*

Remark on the Z

PathnameLookup

LegalPathname
fid! : **FID**

---

fid! = **pathname_fid**(**startfid**?, **pn**?)

The schema **PathnameLookup** *is designed as a data translation schema, for use with pipes ().*
It inherits inputs **startfid**? *and* **pn**? *from* **LegalPathname**.

Remark on the Z

We define **ExistentFile** and **NonExistentFile** as predicate schemas to distinguish whether a (**startfid**?, **pn**?) pair identifies a file in the **FileSpace** state.

ExistentFile

FileSpace
LegalPathnames
startfid? : **FID**
pn? : **PATHNAME**

(**startfid**?, **pn**?) $\in$ dom **pathname_fid**

NonExistentFile

FileSpace
LegalPathnames
startfid? : **FID**
pn? : **PATHNAME**

(**startfid**?, **pn**?) $\notin$ dom **pathname_fid**

We define the schemas **RegularFile** and **DirectoryFile** to be predicates indicating that the file identified by a pathname is a **regular** file or a **directory** file, respectively. These predicates will be used later when we must restrict operations that apply only to one kind of file.

RegularFile

FileSpace
LegalPathnames
startfid? : **FID**
pn? : **PATHNAME**

---

ftype(**pathname_fid**(**startfid**?, **pn**?)) = **regular**

DirectoryFile

FileSpace
LegalPathnames
startfid? : **FID**
pn? : **PATHNAME**

---

ftype(**pathname_fid**(**startfid**?, **pn**?)) = **directory**

EmptyDirectory

DirectoryFile

---

dom(**DirContents**(**pathname_fid**(**startfid**?, **pn**?))) = $\emptyset$

We specify the successful actions of linking, unlinking and creating a file. Pathnames instead of file identifiers are used to name files. We do not specify read, write or destroy at this level, since pathnames are never used in the interfaces to these operations.

The schema **Rootfid_is_preserved** is used in the schemas that are **ΔFileSpace** but do not change **Rootfid**. At the moment, we do not include any operations that change **Rootfid**. However, the Unix system defines the **chroot** operation, which does change the (apparent) root of the file space. So, we provide for that possibility.

Rootfid_is_preserved

ΔFileSpace

Rootfid′ = **Rootfid**

Link

The purpose of a link operation is to associate a new pathname with an existing file.

LinkFS allows us to a create a legal pathname to a file regardless of whether the file is already present in any directory (e.g., a file created by **CreateBFS**). The **pn** is the name of the new link to be installed, and **fid** is its target. **front**(**pn**) must name an existing directory, and **last**(**pn**) must be a name not currently in that directory. **fid** must identify an existing file.

LinkFS

Δ**FileSpace**
LegalPathnames
NonExistentFile
Rootfid_is_preserved
startfid? : **FID**
pn? : **PATHNAME**
fid? : **FID**

---

fid? $\in$ dom **fcontents**

$\#$**pn**? > 0

(**startfid**?, **front**(**pn**?)) $\in$ **legal_pathnames**

(**parentfid** == **pathname_fid**(**startfid**?, **front**(**pn**?)) $\bullet$
  ftype(**parentfid**) = **directory** $\land$
  last(**pn**?) $\notin$ dom(**DirContents**(**parentfid**)) $\land$
  (**dir** == **DirContents**(**parentfid**) $\bullet$
    (**newdir** == **dir** $\cup$ {**last**(**pn**?) $\mapsto$ **fid**?} $\bullet$
      fcontents$'$ = **fcontents** $\oplus$ {**parentfid** $\mapsto$ **UnParseDir**(**newdir**)})))

ftype$'$ = **ftype**

We can describe the new directory contents using $\cup$*, rather than* $\oplus$*, because we know that* **last**(**pn**?) *is not in the domain of the directory.*

Remark on the Z

Unlink

The purpose of an unlink operation is to remove a pathname from the file space. Removing a pathname does not remove it's associated file. Thus we do not remove the corresponding element from either **fcontents** or **ftype**. (At this level files may be left with no links to them. After we introduce the "file table" and "process table", we will add "garbage collection" to the file system.)

The input is the (**startfid**, **pn**) pair which names an existing file. In a legal pathname, all initial subsequences are also legal pathnames, and name directories. So we know, in particular, that (**startfid**?, **front**(**pn**)) identifies a directory in the schema below.

UnlinkFS

Δ**FileSpace**
LegalPathname
ExistentFile
Rootfid_is_preserved
startfid? : **FID**
pn? : **PATHNAME**

$\#\mathbf{pn}? > 0$

$(\mathbf{parentfid} == \mathbf{pathname\_fid}(\mathbf{startfid}?, \mathbf{front}(\mathbf{pn}?)) \bullet$
$\quad (\mathbf{dir} == \mathbf{DirContents}(\mathbf{parentfid}) \bullet$
$\quad\quad (\mathbf{newdir} == \{\mathbf{last}(\mathbf{pn}?)\}\ \mathbf{dir} \bullet$
$\quad\quad\quad \mathbf{fcontents}' = \mathbf{fcontents} \oplus \{\mathbf{parentfid} \mapsto \mathbf{UnParseDir}(\mathbf{newdir})\})))$

$\mathbf{ftype}' = \mathbf{ftype}$

AddFType

The schema **AddFType** is a companion operation to **CreateBFS**. The two operations combine to update **fcontents** and **ftype** when creating a new file.

AddFType

ftype, **ftype**$'$: **FID** $\pfun$ **FTYPE**
fid? : **FID**
ftype? : **FTYPE**

---

fid? $\notin$ dom **ftype**

ftype$'$ = **ftype** $\oplus$ {**fid**? $\mapsto$ **ftype**?}

Remark on the Z

Note that **FileSpace** *includes the property that* dom **ftype** = dom **fcontents**, *which is not a necessary property of AddFType. In fact, AddFType is applied precisely when* **ftype** *does not yet reflect the file type of the new FID* **fid**?. *So, we must combine* **AddFType** *with* **CreateBFS** *so that this invariant holds in both the initial and final states, and then explicitly include* Δ**FileSpace** *to state that the invariant holds.*

Note also that AddFType does not state any relation about the initial and final **Rootfid** *or* **fcontents**. *AddFType will be used in schemas that include statements of those relations. Thus, AddFType is designed as a partial description of a* **FileSpace** *transition, and is intended to be combined with other* Δ**FileSpace** *schemas to more completely describe operations on a* **FileSpace**.

DeleteFType

The operation **DeleteFType** removes the association of a file identifier with an **FTYPE** from the function **ftype**.

DeleteFType

$\mathbf{ftype}, \mathbf{ftype}' : \mathbf{FID}\ \ \mathbf{FTYPE}$
$\mathbf{fid}? : \mathbf{FID}$

$\mathbf{fid}? \in \mathrm{dom}\, \mathbf{ftype}$

$\mathbf{ftype}' = \{\mathbf{fid}?\}\ \mathbf{ftype}$

CreateFS

In a file space, creating a file is a combination of a basic file creation step and a link operation. The **pn** is the name of the new file to be created. The pathname **front**(**pn**) must name an existing directory, and the syllable **last**(**pn**) must be a name not currently in that directory. The input **ftype**? gives the type of the newly created file.

In the schema **CreateFS** these inputs are inherited from the schemas **LinkFS** and **AddFType**. (The input **fid**? to **LinkFS** is captured by **NewFid**, and so is not an input of **CreateFS**.) The file space is changed by the addition of an empty file to the file space as specified by **CreateBFS**, the updating of **ftype** to reflect the new file's type, and the creation of a link to the new file as specified by **LinkFS**.

$$\mathbf{CreateFS} \mathrel{\widehat=} \mathbf{NewFid}((\mathbf{CreateBFS} \land \mathbf{AddFType} \land \mathbf{Rootfid\_is\_preserved})\ \ \mathbf{LinkFS})$$

Note that CreateBFS and AddFType can be combined using $\land$ *because they do not overlap in either their outputs or the state that they "change".*

LinkFS must be added via , because it requires that $\mathbf{fid}? \in \mathrm{dom}\, \mathbf{fcontents}$*, and that is only true for the* $\mathbf{fcontents}'$ *following* **CreateBFS**.

Remark on the Z

---

5 The File Table

A file must be opened before access can occur. Open files are named by elements of the type **OID**.

[**OID**]

A file may be opened in one of several "modes", given permission to do so.

$$\mathbf{ACCESS\_MODE} ::= \mathbf{rdonly} \mid \mathbf{wronly} \mid \mathbf{rdwr} \mid \mathbf{append}$$

The following schema is so named because it models information implemented in the Unix global file table. **ftposn** is the current position for the open file. **ftmode** associates an access mode with the open file. **ftfid** maps an oid to a file identifier.

FileTable

$$\begin{array}{l} \textbf{ftposn} : \textbf{OID} \\ \textbf{ftmode} : \textbf{OID} \;\; \textbf{ACCESS\_MODE} \\ \textbf{ftfid} : \textbf{OID} \;\; \textbf{FID} \\ \hline \mathrm{dom}\, \textbf{ftposn} = \mathrm{dom}\, \textbf{ftmode} = \mathrm{dom}\, \textbf{ftfid} \end{array}$$

We postpone saying $\textbf{ran}(\textbf{ftfid}) \subseteq \mathrm{dom}(\textbf{fcontents})$ *until later, because the* **FileSpace** *schema is not part of the* **FileTable** *schema. This will be introduced in the* **BasicAccessSystem** *(section 8, page 22).*

Remark on the Z

Define some data translation schemas for **OID's**.

OidToFid

$$\begin{array}{l} \textbf{FileTable} \\ \textbf{oid}? : \textbf{OID} \\ \textbf{fid}! : \textbf{FID} \\ \hline \textbf{oid}? \in \mathrm{dom}\, \textbf{ftfid} \\ \textbf{fid}! = \textbf{ftfid}(\textbf{oid}?) \end{array}$$

OidToPosn

$$\begin{array}{l} \textbf{FileTable} \\ \textbf{oid}? : \textbf{OID} \\ \textbf{offset}! : \\ \hline \textbf{oid}? \in \mathrm{dom}\, \textbf{ftposn} \\ \textbf{offset}! = \textbf{ftposn}(\textbf{oid}?) \end{array}$$

Open

The schema **OpenFT** specifies a successful file open operation. The file to be opened is specified by **fid**?. **Oid**! is a new open file identifier. The input **access_mode**? specifies the access mode to the open file. The initial position of the opened file is zero.

Directories are updated via the **LinkFS** and **UnlinkFS** operations. The restriction that directory files only be opened in **rdonly** mode is not imposed until later (see section 8, The Basic Access System), because **FileTable** does not include the schema **FileSpace**, and so does not include **ftype**.

When a file is opened for writing (but not for appending), the file is truncated to zero length. This corresponds to requiring the `O_TRUNC` option of the Unix `open` call. It would be easy to extend the specification

to include opening for write without the `O_TRUNC` option, as well as some additional `open` options (e.g., create, exclusive create).

However, some options, such as `O_SYNC`, are not so simple to include. The `O_SYNC` option requires that there be no output buffering on write operations (i.e., that the `write` call not complete until the required physical I/O has completed.) However, the file system specification does not describe buffered I/O; it describes the required changes in the state of the file space. Buffered I/O is an optimization strategy in the implementation. Optimizations should not alter the semantics of an operation, only its performance. A commonly accepted requirement for an optimization is that it not change the semantics of a successful operation, but that it may change the semantics of operations that fail (e.g., a different error may be reported than in the unoptimized implementation). However, we observe that the Unix file system admits buffered I/O in its fundamental file I/O model, and so includes options for dealing with the effects of buffering on file operations. We have not included buffered I/O in our specification of SFS.

If a file is open for reading at the time that it is opened for writing, then this truncation can be observed. There are other choices for the meaning of opening a file for writing when it is already open for reading. Other choices may make the file system specification more complex. We have chosen one that seems particularly simple, although it does not model all of the rich, complex behavior of the Unix file system.

OpenFT *illustrates a pattern for defining a schema that operates on a portion of its state.* **OpenFT** *operates on the* **FileTable**, *but also needs to refer to the state of the* **BasicFileSpace**. *We include the declaration* Δ**FileTable** *to indicate the former. But we include* **BasicFileSpace** *for the latter, rather than* Ξ**BasicFileSpace**. *We do this because we may combine* **OpenFT** *with other schemas that perform corresponding operations on the* **BasicFileSpace** *using the pipe operator (). If we included* Ξ**BasicFileSpace** *here, then* **OpenFT** *would include the property that* $\mathbf{fcontents}' = \mathbf{fcontents}$. *Including that property would prevent us from piping between* **OpenFT** *and a schema that operates on* **BasicFileSpace**.

Remark on the Z

OpenInternalFT

Δ**FileTable**
BasicFileSpace
$\mathbf{fid}? : \mathbf{FID}$
$\mathbf{access\_mode}? : \mathbf{ACCESS\_MODE}$
$\mathbf{oid}! : \mathbf{OID}$

$\mathbf{fid}? \in \operatorname{dom} \mathbf{fcontents}$

$\mathbf{oid}! \notin \operatorname{dom} \mathbf{ftfid}$

$\mathbf{ftfid}' = \mathbf{ftfid} \oplus \{\mathbf{oid}! \mapsto \mathbf{fid}?\}$

$\mathbf{ftmode}' = \mathbf{ftmode} \oplus \{\mathbf{oid}! \mapsto \mathbf{access\_mode}?\}$

$\mathbf{ftposn}' = \mathbf{ftposn} \oplus \{\mathbf{oid}! \mapsto 0\}$

TruncateFile

Δ**BasicFileSpace**
$\mathbf{fid}? : \mathbf{FID}$

$\mathbf{fcontents}' = \mathbf{fcontents} \oplus \{\mathbf{fid}? \mapsto \langle\rangle\}$

TruncatingOpen

$$\textbf{access\_mode}? : \textbf{ACCESS\_MODE}$$

$$\textbf{access\_mode}? \in \{\textbf{wronly}, \textbf{rdwr}\}$$

$$\textbf{MaybeTruncateFile} \mathrel{\widehat=} \textbf{TruncatingOpen} \Rightarrow \textbf{TruncateFile}$$

$$\textbf{OpenFT} \mathrel{\widehat=} \textbf{OpenInternalFT} \land \textbf{MaybeTruncateFile}$$

CloseFT

We specify the operation of destroying a file table entry. This is the opposite of **OpenFT**. Note, that this effect does not always occur when closing a file at the Unix File System level; a file table entry is expunged only when no process references it. Since a file table entry may be shared by two process file tables, or (in Unix, but not in this draft specification) by multiple entries in a single process table.

CloseFT

$$\Delta\textbf{FileTable}$$
$$\textbf{oid}? : \textbf{OID}$$

$$\textbf{oid}? \in \operatorname{dom} \textbf{ftfid}$$
$$\textbf{ftfid}' = \{\textbf{oid}?\} \mathbin{\lhd} \textbf{ftfid}$$
$$\textbf{ftmode}' = \{\textbf{oid}?\} \mathbin{\lhd} \textbf{ftmode}$$
$$\textbf{ftposn}' = \{\textbf{oid}?\} \mathbin{\lhd} \textbf{ftposn}$$

The property $\textbf{oid} \in \operatorname{dom} \textbf{ftfid}$ is not necessary in this schema. However, it does indicate our intention about the use of this operation. The **CloseFT** operation could be extended to operate on oid's that do not have entries the file table, in which case the file table should remain unchanged[7].

Seek

The schema **SeekFT** specifies the operation of logically updating an open file's position. No data is read or written. **SeekFT** merely updates the **ftposn** entry in the file table. Thus, you can seek to a position outside of the current file without error.

[7] The expression $\{\textbf{oid}?\} \mathbin{\lhd} \textbf{ftfid}$ yields **ftfid** if $\textbf{oid}? \notin \operatorname{dom} \textbf{ftfid}$. So we need only remove the restriction $\textbf{oid}? \in \operatorname{dom} \textbf{ftfid}$ from the current schema to effect this change.

SeekFT

Δ**FileTable**
oid? : **OID**
offset? :

---

oid? $\in$ dom **ftfid**

ftposn$'$ = **ftposn** $\oplus$ {**oid**? $\mapsto$ **offset**?}

ftmode$'$ = **ftmode**

ftfid$'$ = **ftfid**

The schema **IncrPosnFT** increments the open file position by a specified amount (**delta**?). This will be used later to describe updating the file position after a read or write operation.

IncrPosnFT

Δ**FileTable**
oid? : **OID**
delta? :

---

oid? $\in$ dom **ftfid**

ftposn$'$ = **ftposn** $\oplus$ {**oid**? $\mapsto$ **ftposn**(**oid**?) + **delta**?}

ftmode$'$ = **ftmode**

ftfid$'$ = **ftfid**

SeekEOF $\widehat{=}$ **OidToFidFileLengthSeekFT**

We now define three more schemas to be used as predicates. They allow us to distinguish classes of access modes recorded in the **FileTable**.

Opened For Read

OpenedForRead

FileTable
oid? : **OID**

---

ftmode(**oid**?) $\in$ {**rdonly**, **rdwr**}

Opened For Write

OpenedForWrite

FileTable
oid? : **OID**

---

ftmode(**oid**?) $\in$ {**wronly**, **rdwr**}

Opened For Append

OpenedForAppend

FileTable
oid? : **OID**

---

ftmode(**oid**?) = **append**

6 Processes

The system is populated with processes, each identified by an element of the set **PID**. Each process has an associated file descriptor table, a mapping from file descriptors (type **FD**) to open file identifiers. Each process has its own space of file descriptors, through which it can reference file table entries.

[**PID**, **FD**]

ProcessTable

pfdtable : **PID** (**FD** **OID**)

FdToOid

The schema **FdToOid** translates a pair of inputs (**pid**? and **fd**?), which identify an entry in a **pfdtable**, into the **OID** that entry points to.

FdToOid

ProcessTable
pid? : **PID**
fd? : **FD**
oid! : **OID**

---

pid? ∈ dom **pfdtable**

fd? ∈ dom(**pfdtable**(**pid**?))

oid! = (**pfdtable**(**pid**?))(**fd**?)

FdToFid ≙ **FdToOidOidToFid**

There are four operations on the **ProcessTable**:

- add a new file-descriptor entry,
- delete an existing file-descriptor entry,
- fork a new process, which inherits its initial **pfdtable** from its parent, and
- duplicate a **pfdtable** entry within that **pfdtable**.

AddProcessFD

AddProcessFD

$$\Delta \mathbf{ProcessTable}$$
$$\mathbf{pid}? : \mathbf{PID}$$
$$\mathbf{oid}? : \mathbf{OID}$$
$$\mathbf{fd}! : \mathbf{FD}$$

$$\mathbf{pid}? \in \operatorname{dom} \mathbf{pfdtable}$$
$$\mathbf{fd}! \notin \operatorname{dom}(\mathbf{pfdtable}(\mathbf{pid}?))$$
$$(\mathbf{newfdtable} == (\mathbf{pfdtable}(\mathbf{pid}?)) \oplus \{\mathbf{fd}! \mapsto \mathbf{oid}?\} \bullet \mathbf{pfdtable}' = \mathbf{pfdtable} \oplus \{\mathbf{pid}? \mapsto \mathbf{newfdtable}\})$$

DeleteProcessFD

DeleteProcessFD

$$\Delta \mathbf{ProcessTable}$$
$$\mathbf{pid}? : \mathbf{PID}$$
$$\mathbf{fd}? : \mathbf{FD}$$

$$\mathbf{pid}? \in \operatorname{dom} \mathbf{pfdtable}$$
$$\mathbf{fd}? \in \operatorname{dom}(\mathbf{pfdtable}(\mathbf{pid}?))$$
$$(\mathbf{newfdtable} == \{\mathbf{fd}?\}\ (\mathbf{pfdtable}(\mathbf{pid}?)) \bullet \mathbf{pfdtable}' = \mathbf{pfdtable} \oplus \{\mathbf{pid}? \mapsto \mathbf{newfdtable}\})$$

ForkProcess

We partially specify the Unix fork operation. A fork creates a new process, which inherits the file descriptor table of its parent process. The input **pid**? identifies the parent process. The output **pid**! identifies the child process.

ForkPT

$$\Delta \mathbf{ProcessTable}$$
$$\mathbf{pid?} : \mathbf{PID}$$
$$\mathbf{pid!} : \mathbf{PID}$$

$$\mathbf{pid?} \in \mathrm{dom}\, \mathbf{pfdtable}$$
$$\mathbf{pid!} \notin \mathrm{dom}\, \mathbf{pfdtable}$$
$$\mathbf{pfdtable}' = \mathbf{pfdtable} \oplus \{\mathbf{pid!} \mapsto \mathbf{pfdtable}(\mathbf{pid?})\}$$

DuplicateFD

DuplicateFD

$$\Delta \mathbf{ProcessTable}$$
$$\mathbf{pid?} : \mathbf{PID}$$
$$\mathbf{fd?} : \mathbf{FD}$$
$$\mathbf{fd!} : \mathbf{FD}$$

$$\mathbf{pid?} \in \mathrm{dom}\, \mathbf{pfdtable}$$
$$\mathbf{fd?} \in \mathrm{dom}(\mathbf{pfdtable}(\mathbf{pid?}))$$
$$\mathbf{fd!} \notin \mathrm{dom}(\mathbf{pfdtable}(\mathbf{pid?}))$$
$$(\mathbf{oid} == (\mathbf{pfdtable}(\mathbf{pid?}))(\mathbf{fd?}) \bullet$$
$$\quad \mathbf{newfdtable} == (\mathbf{pfdtable}(\mathbf{pid?})) \oplus \{\mathbf{fd!} \mapsto \mathbf{oid}\} \bullet$$
$$\quad\quad \mathbf{pfdtable}' = \mathbf{pfdtable} \oplus \{\mathbf{pid?} \mapsto \mathbf{newfdtable}\})$$

7 Garbage Collection

In a system state, a file may be referenced in several ways: it may have one or more links from directories, and it may be open to one or more processes. A file's contents are removed from the system only when all references to it disappear.[8] Here we specify garbage collection by directly describing the presence or absence of **ProcessTable** and directory references.

ExistsLinkReferenceToFid

$$\mathbf{FileSpace}$$
$$\mathbf{LegalPathnames}$$
$$\mathbf{fid?} : \mathbf{FID}$$

$$\exists\, \mathbf{pn} : \mathbf{PATHNAME} \bullet$$
$$\quad ((\mathbf{Rootfid}, \mathbf{pn}) \in \mathbf{legal\_pathnames} \land$$
$$\quad\quad \mathbf{pathname\_fid}(\mathbf{Rootfid}, \mathbf{pn}) = \mathbf{fid?})$$

[8] In the standard Unix implementation, each inode contains a count of all the links referencing it, and each file table entry contains a count of the file descriptors in the process file-tables that reference it. When the reference count of an inode is zero, there are no remaining link references in the directories of the file system. When the reference count of a file-table entry is zero, there are no remaining file descriptor entries in any process' file-table referring to that file-table entry. The Unix file system "deletes" a file (as opposed to merely removing a directory link to a file) when both of these counts are zero.

ExistsFileTableReferenceToFid

FileTable
fid? : **FID**

$\exists$ **oid** : **OID** | **oid** $\in$ dom **ftfid** $\bullet$ **ftfid**(**oid**) = **fid**?

ExistsReferenceToFid $\widehat{=}$ **ExistsLinkReferenceToFid** $\vee$ **ExistsFileTableReferenceToFid**

A file may be destroyed when there are no references to it. **FileSpaceGC** specifies that a file is garbage collected if there are no references to it, otherwise there is no state change. This is a Δ**FileSpace** transition, but it also depends on **FileTable**; so we declare it as Ξ**FileTable** to indicate that it does not affect the **FileTable** state (and also to state that the **FileTable** variables are preserved as outputs of **FileSpaceGC**).

FileSpaceGC $\widehat{=}$ ($\neg$ **ExistsReferenceToFid** $\Leftrightarrow$ (**DestroyBFS** $\wedge$ **DeleteFType**
$\wedge$ **Rootfid_is_preserved** $\wedge$ Δ**FileSpace**))
$\wedge$ (**ExistsReferenceToFid** $\Leftrightarrow$ Ξ**FileSpace**)
$\wedge$ Ξ**FileTable**

FileTableGC specifies that a file table entry is garbage collected if there are no references to it, otherwise there is no state change. This transition does not depend on the **FileSpace** state, so we need not include Ξ**FileSpace** here.

ExistsReferenceToFileTableEntry

ProcessTable
oid? : **OID**

$\exists$ **pid** : dom **pfdtable** $\bullet$ $\exists$ **fd** : dom(**pfdtable**(**pid**)) $\bullet$
(**pfdtable**(**pid**))(**fd**) = **oid**?

FileTableGC $\widehat{=}$ ($\neg$ **ExistsReferenceToFileTableEntry** $\Leftrightarrow$ **CloseFT**)
$\wedge$ (**ExistsReferenceToFileTableEntry** $\Leftrightarrow$ Ξ**FileTable**)

8 The Basic Access System

The state of the Basic Access System includes the file space, the file table and the process table. A **BasicAccessSystem** has several additional properties compared to a **FileSpace**.

- File table entries must point to files in the **FileSpace**.
- Process file descriptor table entries must point to file table entries. (That is, every process **fd** must map to a file table entry.)

- Every file table entry is pointed to by some process file descriptor table entry. (That is, every file table entry is the image of some process' file descriptor.)
- Every file in the **FileSpace** must either be reachable from the root or be currently open.

The second and third properties together state that all **oid**'s accessible from the **pfdtable** occur in the file table, and all **oid**'s in the file table also occur somewhere in the **pfdtable**. This can be stated succinctly in Z:

$$\mathrm{dom}\,\mathbf{ftfid} = \bigcup\{\ \mathbf{pid} : \mathrm{dom}\,\mathbf{pfdtable} \bullet \mathrm{ran}(\mathbf{pfdtable}(\mathbf{pid}))\ \}$$

This says that the domain of **ftfid** (i.e., all open file **oid**$'$**s**) is the same as the union of the ranges of all process file tables (i.e., all of the **oid**$'$**s** reachable from some **fd** in the process table for some **pid**).

These two properties require garbage collection of inaccessible entries in the **FileTable**. The fourth property additionally requires garbage collection of the **FileSpace**. To state it we introduce the function **rooted_pathname_fid**, which is simply **pathname_fid** with the restriction that the starting **FID** be the **Rootfid** of the **FileSpace**.

RootedPathnames

$$\begin{array}{l}\mathbf{LegalPathnames}\\ \mathbf{rooted\_pathname\_fid} : \mathbf{PATHNAME}\ \ \mathbf{FID}\\ \hline \forall\, \mathbf{pn} : \mathbf{PATHNAME} \bullet\\ \quad \mathbf{rooted\_pathname\_fid}(\mathbf{pn}) = \mathbf{pathname\_fid}(\mathbf{Rootfid}, \mathbf{pn})\end{array}$$

BasicAccessSystem

$$\begin{array}{l}\mathbf{FileSpace}\\ \mathbf{LegalPathnames}\\ \mathbf{RootedPathnames}\\ \mathbf{FileTable}\\ \mathbf{ProcessTable}\\ \hline \mathrm{ran}\,\mathbf{ftfid} \subseteq \mathrm{dom}\,\mathbf{fcontents}\\ \mathrm{dom}\,\mathbf{ftfid} = \bigcup\{\ \mathbf{pid} : \mathrm{dom}\,\mathbf{pfdtable} \bullet \mathrm{ran}(\mathbf{pfdtable}(\mathbf{pid}))\ \}\\ \forall\, \mathbf{fid} : \mathrm{dom}\,\mathbf{fcontents} \bullet\\ \qquad \mathbf{fid} \in \mathrm{ran}\,\mathbf{rooted\_pathname\_fid} \lor\\ \qquad \mathbf{fid} \in \mathrm{ran}\,\mathbf{ftfid}\end{array}$$

Open

Open updates the file table and process table in the basic access system, but not the file space. The file table update is described in the schema **OpenFT**. The process table is updated to map a new file descriptor to a new file table entry. The file must already exist. The **Create** operation is used to create a new file.

Directories are updated only by the **LinkBAS** and **UnlinkBAS** operations. Directory files cannot be changed via the write operation. This restriction is imposed by including the schema **OnlyOpenDirectoriesRDONLY** in **OpenBAS**.

OnlyOpenDirectoriesRDONLY

FileSpace
fid? : **FID**
access_mode? : **ACCESS_MODE**

---

fid? ∈ dom **ftype**

ftype(**fid**?) = **directory** ⇒ **access_mode**? = **rdonly**

OpenBAS ≙ **PathnameLookup**
(**OnlyOpenDirectoriesRDONLY** ∧ **OpenFT**)
AddProcessFD

If our model were to stop at the **BAS** *level, we would include* Ξ**FileSpace** *in the definition of* **OpenBAS**, *since an open operation does not change the* **FileSpace**. *However, we are building toward a model of* **UFS** *and* **SFS**. *So we will leave out the* Ξ**FileSpace**, *so that later at the* **UFS** *level we can pipe from* **CreateBAS**, *which does change the* **FileSpace**, *to* **OpenBAS**. *Similarly, we will not include* Ξ**FileTable** *in the definition of* **CreateBAS**.

Remark on the Z

Close

CloseBAS ≙ (**FdToOid** ∧ **FdToFid**)
(**DeleteProcessFD**
FileTableGC
(**FileSpaceGC** ∧ Ξ**FileTable**))

Without the Ξ**FileTable**, *gives the signature of* **CloseBAS** *as excluding the* **FileTable**′ *state variables* **ftfid**′ *and friends. The maps incoming* **ftfid**′ *to* **ftfid**. *Even though* **FileTableGC** *is declared* Δ**FileTable**, *apparently the*

...**FileSpaceGC** *eats up the* **FileTable**′ *and, since* **FileSpaceGC** isn't Δ**FileTable**, *it doesn't produce a new set of "final state" variables.*

Remark on the Z

Link

The **LinkBAS** schema takes the same logical inputs as **LinkFS**, but the names are different. The existing file (the target of the link) is identified by the input pair (**xstartfid**?, **xpn**?), rather than an **fid**. The input pair (**startfid**?, **newpn**?), inherited from **LinkBAS**, still identifies the new pathname. **LinkBAS**

LinkBAS ≙ (**PathnameLookup**[**xstartfid**?/**startfid**?, **xpn**?/**pn**?]
LinkFS)
∧ Ξ**FileTable** ∧ Ξ**ProcessTable**

Unlink

The file to be unlinked is identified by (**startfid**?, **pn**?). That file must exist and be either a regular file or an empty directory.

The prohibition against unlinking non-empty directories simplifies the problem of garbage collecting the file space, because only the file being unlinked is a candidate for garbage collection. If unlinking non-empty directories were permitted, then the entire tree of files under the newly unlinked directory would have to be considered.

$$\mathbf{UnlinkBAS} \mathrel{\widehat=} \mathbf{PathnameLookup} \\ ((\mathbf{RegularFile} \lor \mathbf{EmptyDirectory}) \land \mathbf{UnlinkFS}\ \ \mathbf{FileSpaceGC})$$

Create

The **BasicAccessSystem** added **FileTable** and **ProcessTable** to the **FileSpace**. Since creating a file affects neither the **FileTable** nor the **ProcessTable**, **CreateBAS** need add nothing to **CreateFS**. It takes the same inputs (**startfid**?, **pn**?) and **ftype**? and has signature Δ**FileSpace**.

$$\mathbf{CreateBAS} \mathrel{\widehat=} \mathbf{CreateFS}$$

Read

The file to be read is specified by a file descriptor (**FD**). The file must already be opened in one of the access modes that allows a read operation. The open file's position is incremented by the amount of data read.

DataLength

$$\mathbf{data}? : \mathbf{FILE}$$
$$\mathbf{data}! : \mathbf{FILE}$$
$$\mathbf{delta}! :$$

$$\mathbf{delta}! = \#\mathbf{data}?$$
$$\mathbf{data}! = \mathbf{data}?$$

$$\mathbf{ReadBAS} \mathrel{\widehat=} \mathbf{FdToOid} \\ (\mathbf{OidToFid} \land \mathbf{OpenedForRead} \land \mathbf{OidToPosn}) \\ \mathbf{ReadBFSDataLength}(\mathbf{FdToOidIncrPosnFT})$$

Write

The file to be written is specified by a file descriptor (**FD**). The file must already be opened in one of the access modes that allows a write operation. The offset where writing begins is given by the file table

position, and therefore this input parameter is hidden in the ultimate specification for **WriteBAS**. The file table state is updated to reflect the number of bytes written.

Note that the behavior of writing to a file when it is opened for *write* differs from when it is opened for *append*. When opened for write, the data is written at the current file position (as represented in **ftposn** in **FileTable**). When opened for append, the file position is always set to the end of the file before the data is written. In both cases the file position is incremented by the length of the data written.

The *write* operation is described just below in the **WriteBAS** schema; the *append* operation is described separately in the **AppendBAS** schema.

$$\begin{aligned}&\mathbf{WriteBAS} \mathrel{\widehat=} \mathbf{FdToOid}\\ &\qquad (\mathbf{OidToFid} \land (\mathbf{OpenedForWrite} \lor \mathbf{OpenedForAppend}) \land \mathbf{OidToPosn})\\ &\qquad \mathbf{WriteBFS}\\ &\qquad \mathbf{DataLength} \setminus (\mathbf{data}!)\\ &\qquad (\mathbf{FdToOidIncrPosnFT})\end{aligned}$$

Remark on the Z

Observe that in **WriteBAS** *the schema* **DataLength** *gets its input* **data**? *from the outer-most inputs.* **WriteBFS** *doesn't capture it, because it's not an output being fed down the pipe. But in* **ReadBAS**, **DataLength** *does capture the* **data**! *output from* **ReadBFS**, *and so it must propagate it to the next stage of the pipe. Since we don't want* **data**! *as an output of* **WriteBAS**, *we hide it as an output of* **DataLength**.

Append

The append operation is similar to the write operation, but always positions the file at EOF before the data is written.

$$\mathbf{AppendBAS} \mathrel{\widehat=} (\mathbf{FdToOidSeekEOF}) \;\; \mathbf{WriteBAS}$$

Remark on the Z

Note that the **FdToOid** *eats the* **fd**? *input and produces the* **oid**? *input required by* **SeekEOF**. *But the semicolon operator () sees the original* **fd**? *from the input arguments to the top-level schema-expression, which is required as input to* **WriteBAS**.

Fork

Fork describes the effect on the file system of a forking a new process. The new process has its own process file table, which is initially a copy of the file table of its parent process. This operation creates a new *pid* and adds the appropriate entry to the process table. The work is done by **ForkPT**.

ForkBAS

$$\begin{aligned}&\Delta\mathbf{BasicAccessSystem}\\ &\Xi\mathbf{FileTable}\\ &\Xi\mathbf{FileSpace}\\ &\mathbf{ForkPT}\end{aligned}$$

ForkBAS *is* Ξ**FileTable** *because we don't include reference counts in the file table model. If you wanted to add reference counts, that could be done by defining a new schema (***FileTableRefCnt***) with the function* **ftrefcnt** : **OID** . *Then* **ForkBAS** *would be* Δ**FileTableRefCnt**, *but still* Ξ**FileTable**.

Remark on the Z

---

9 Discretionary Access Control

Our basic model of access control revolves around a set of permissions. The access control policy provides the *current* set of permissions given an operation and the policy-relevant attributes of the operands. Each operation has a set of permissions defined as *necessary* for it to complete. The definition of the operation, in turn, must test whether this set of necessary permissions is included in the set of current permissions provided by the access control policy. The defined set of permissions is the language with which the policy and the enforcement mechanism communicate.

At the **DAC** level we recognize three types of access permission to a file: **r**, **w**, and **x**. Processes and files have the attributes that are used to determine discretionary access. For this DAC model a file has an *owner* and a *group*; a process has an *owner* and a *set of groups*. The types **USER** and **GROUP** are used to model the ownership and group-membership of a file or a process.[9]

We do not currently model execution. So the **x** permission is only used as "search" permission for a directory, which permits a search for a specific name in the directory. **x** permission does not permit listing all of the names in the directory or reading the directory.

We do not model the "setuid", "setgid", or the "sticky" permissions of the full Unix file system.

$$\mathbf{PERM} ::= \mathbf{r} \mid \mathbf{w} \mid \mathbf{x}$$

$$\mathbf{PERMS} == (\mathbf{PERM})$$

$$[\mathbf{USER}, \mathbf{GROUP}]$$

For convenience we define some "record structures" to contain related variables. The record schema **FPERMS** contains the three permission fields named **owner_perms**, **group_perms**, and **other_perms**.

FPERMS

$$\mathbf{owner\_perms}, \mathbf{group\_perms}, \mathbf{other\_perms} : \mathbf{PERMS}$$

DAC_File_Attributes

$$\mathbf{owner} : \mathbf{USER}$$
$$\mathbf{group} : \mathbf{GROUP}$$
$$\mathbf{fperms} : \mathbf{FPERMS}$$

[9] Our model is BSD-like, in this respect, associating a process with a set of groups, rather than a single group as for System V-like systems. We do not model the Unix notions of "real gid" and "effective gid".

DAC_Process_Attributes

$$\mathbf{owner} : \mathbf{USER}$$
$$\mathbf{groups} : \mathbf{GROUP}$$

The type **DACops** names the operations that to which the discretionary access will apply. Note that we have introduced the **search** operation here. Now that directories have permissions associated with them, a process may be prohibited from searching for a name in a directory. Directory permissions may permit searching for a name in a directory but not reading the directory. That is, a process may be allowed to search for a specific name in a directory, but not read the directory to learn all of the names in it. The **x** file permission is interpreted as the **search** permission for a directory.

$$\mathbf{DACops} ::= \mathbf{open} \mid \mathbf{link} \mid \mathbf{unlink} \mid \mathbf{create} \mid \mathbf{search}$$

DAC

$$\mathbf{fdac} : \mathbf{FID}\ \ \mathbf{DAC\_File\_Attributes}$$
$$\mathbf{pdac} : \mathbf{PID}\ \ \mathbf{DAC\_Process\_Attributes}$$
$$\mathbf{DAC\_current\_perms} : \mathbf{PID} \times \mathbf{FID}\ \ \mathbf{PERMS}$$
$$\mathbf{DAC\_necessary\_perms} : \mathbf{DACops} \times \mathbf{ACCESS\_MODE}\ \ \mathbf{PERMS}$$

$$\forall \mathbf{pid} : \mathbf{PID};\ \ \mathbf{fid} : \mathbf{FID} \bullet$$
$$\mathbf{DAC\_current\_perms}(\mathbf{pid}, \mathbf{fid}) =$$
$$(\mathbf{pdac\ pid}).\mathbf{owner} = (\mathbf{fdac\ fid}).\mathbf{owner}$$
$$(\mathbf{fdac\ fid}).\mathbf{fperms}.\mathbf{owner\_perms}$$
$$(\mathbf{fdac\ fid}).\mathbf{group} \in (\mathbf{pdac\ pid}).\mathbf{groups}$$
$$(\mathbf{fdac\ fid}).\mathbf{fperms}.\mathbf{group\_perms}$$
$$(\mathbf{fdac\ fid}).\mathbf{fperms}.\mathbf{other\_perms}$$

$$\mathbf{DAC\_necessary\_perms}(\mathbf{open}, \mathbf{rdonly}) = \{\mathbf{r}\}$$

$$\mathbf{DAC\_necessary\_perms}(\mathbf{open}, \mathbf{rdwr}) = \{\mathbf{r}, \mathbf{w}\}$$

$$\mathbf{DAC\_necessary\_perms}(\mathbf{open}, \mathbf{wronly}) = \{\mathbf{w}\}$$

$$\mathbf{DAC\_necessary\_perms}(\mathbf{open}, \mathbf{append}) = \{\mathbf{w}\}$$

$$\forall \mathbf{access\_mode} : \mathbf{ACCESS\_MODE} \bullet$$
$$\mathbf{DAC\_necessary\_perms}(\mathbf{create}, \mathbf{access\_mode}) = \{\mathbf{x}, \mathbf{w}\} \wedge$$
$$\mathbf{DAC\_necessary\_perms}(\mathbf{link}, \mathbf{access\_mode}) = \{\mathbf{x}, \mathbf{w}\} \wedge$$
$$\mathbf{DAC\_necessary\_perms}(\mathbf{unlink}, \mathbf{access\_mode}) = \{\mathbf{x}, \mathbf{w}\} \wedge$$
$$\mathbf{DAC\_necessary\_perms}(\mathbf{create}, \mathbf{access\_mode}) = \{\mathbf{x}, \mathbf{w}\} \wedge$$
$$\mathbf{DAC\_necessary\_perms}(\mathbf{search}, \mathbf{access\_mode}) = \{\mathbf{x}\}$$

Remark on the Z

The schema **DAC** *declares the functions* **fdac** *and* **pdac** *that map an* **fid** *or a* **pid** *to some attributes. The schema* **DAC_File_Attributes** *(in effect) defines a record structure to represent these attributes (***fowner***, and* **fgroup***).* **DAC_Process_Attributes** *defines an analogous structure for processes.*

The operation **DestroyFileAttrDAC** is never invoked directly. It is only invoked implicitly by **UnlinkOK_UFS**.

DestroyFileAttrDAC

$$
\begin{array}{l}
\Delta \mathbf{DAC} \\
\mathbf{BasicAccessSystem} \\
\mathbf{LegalPathname} \\
\mathbf{pid?} : \mathbf{PID} \\
\mathbf{startfid?} : \mathbf{FID} \\
\mathbf{pn?} : \mathbf{PATHNAME} \\
\hline
\mathbf{fid} == \mathbf{pathname\_fid}(\mathbf{startfid?}, \mathbf{pn?}) \bullet \\
\quad \mathbf{fdac}' = \{\mathbf{fid}\} \ \mathbf{fdac} \\
\mathbf{pdac}' = \mathbf{pdac}
\end{array}
$$

The schema **DACVisiblePathnames** defines what pathnames a process can utilize in DAC file system operations. If a pathname is not "visible" then the user program cannot use that name. Note that visibility is a property of a pathname, not of a file. If a file has several links to it, each link has a different pathname. Some of these pathnames may be visible to a given process and some may not be. Note that if a pathname is in $\mathbf{DACvisible\_pathnames}(\mathbf{pid}, \mathbf{fid})$, then it must also be a legal pathname.

Visible pathnames are described as a function from $\mathbf{PID} \times \mathbf{FID}$. The **FID** represents the starting directory for resolving the pathname. At the user level this will be either the root fid or the fid of the process' current working directory. If the process does not have search access to a directory, then no pathname starting from that directory will be visible. Thus, a process will not be able to refer to absolute if it does not have search access to the root directory, and will not be able to refer to relative pathnames if it does not have search access to its current working directory.

DACVisiblePathnames

$$
\begin{array}{l}
\mathbf{FileSpace} \\
\mathbf{LegalPathnames} \\
\mathbf{DAC} \\
\mathbf{DACvisible\_pathnames} : \mathbf{PID} \times \mathbf{FID} \rightarrow (\mathbf{PATHNAME}) \\
\hline
\forall \mathbf{pid} : \mathbf{PID};\ \mathbf{fid} : \mathbf{FID};\ \mathbf{pn} : \mathbf{PATHNAME} \bullet \\
\qquad \langle\rangle \in \mathbf{DACvisible\_pathnames}(\mathbf{pid}, \mathbf{fid}) \\
\qquad \Leftrightarrow \mathbf{DAC\_necessary\_perms}(\mathbf{search}, \mathbf{rdonly}) \subseteq \mathbf{DAC\_current\_perms}(\mathbf{pid}, \mathbf{fid}) \\
\quad \wedge \\
\qquad (\#\mathbf{pn} > 0 \\
\qquad\quad \wedge\ \mathbf{pn} \in \mathbf{DACvisible\_pathnames}(\mathbf{pid}, \mathbf{fid})) \\
\qquad \Leftrightarrow \mathbf{front}(\mathbf{pn}) \in \mathbf{DACvisible\_pathnames}(\mathbf{pid}, \mathbf{fid}) \\
\qquad\quad \wedge\ (\mathbf{fid}, \mathbf{pn}) \in \mathbf{legal\_pathnames} \\
\qquad\quad \wedge\ \mathbf{DAC\_necessary\_perms}(\mathbf{search}, \mathbf{rdonly}) \\
\qquad\quad\quad \subseteq \mathbf{DAC\_current\_perms}(\mathbf{pid}, \mathbf{pathname\_fid}(\mathbf{fid}, (\mathbf{front}(\mathbf{pn}))))
\end{array}
$$

DACVisiblePathname

DACVisiblePathnames
pid? : **PID**
startfid? : **FID**
pn? : **PATHNAME**

---

pn? ∈ **DACvisible_pathnames**(**pid**?, **startfid**?)

The **DAC** test for each operation is basically

DAC_necessary_perms(**Operation**, **Access_Mode**) ⊆ **DAC_current_perms**(**Pid**, **Fid**)

Operations will also require that their pathname inputs be appropriately visible. (E.g., for the link operation, the **front** of the pathname to be created must exist, but the full pathname must not exist.)

Remark on the Z

Since we defined **DAC_necessary_perms** *to take both a* **DACop** *and an* **access_mode** *as arguments, we must associate an* **access_mode** *with each operation when checking* **DAC** *permissions. Since* **Link** *and* **Unlink** *do not normally have any associated access modes, we adopt the arbitrary convention of associating them with* **rdwr** *mode.*

10 The Unix File System Interface

The UFS level combines the **BasicAccessSystem** with the **DAC**. In addition, it restricts some operations according to file type. A number of these restrictions appear to be a mechanism to enforce a protocol on file manipulation on processes so that desirable UFS invariants can be enforced. For example, there can only be a single link to a directory file. (This means that the file space underlying the UFS is restricted to being a rooted tree, instead of an arbitrary graph.)

At the UFS level, **link** and **unlink** are only permitted on regular files, not on directories. Only one link to a directory is permitted, the one created when the directory is created. That link to a directory is removed by the **rmdir** operation, which is just like **UnlinkFS**, but requires additionally that the directory be empty. Also, rather than a single **create** operation for both files and directories, **create** is restricted to creating **regular** files and the new operation **mkdir** is used to create **directory** files. The **write** operation is also not permitted on directories; directories are only updated by **link**, **unlink**, **create**, **mkdir**, and **rmdir**.

The "current process" is represented by the current process identifier, **cpid**. Each process has a "working directory". This will used as the starting **FID** when resolving "relative pathnames." The mapping from each **pid** to its working directory is represented in the state variable **pcwd**. Note that there is no constraint that ran **pcwd** ⊆ dom **fcontents**! There is no mechanism to prevent a process' from deleting (i.e., unlinking) a directory that is in use as a working directory. Unix permits this, and so we do, as well.[10] Further, user process need not have read/search access to either the root directory or to its current working directory. Of course, in that case the process will not be able to reference absolute or relative pathnames (respectively).

[10] If **chmod** were included in our model, we would require that the new working directory exist – and be a directory – in order for the **chmod** operation to be successful.

UFS

BasicAccessSystem
DAC
LegalPathnames
pcwd : **PID** $\rightarrow$ **FID**
cpid : **PID**

---

dom **fdac** = dom **fcontents**

dom **pdac** = dom **pfdtable**

dom **pcwd** = dom **pfdtable**

ran **pcwd** $\subseteq$ **directory_fids**

cpid $\in$ dom **pfdtable**

The transitions on the controlled access system are those of the basic access system, with additional access control constraints. At this level, pathnames are labeled either relative or absolute.

PATHNAMETYPE ::= **relative** | **absolute**

LABELED_PATHNAME == **PATHNAMETYPE** $\times$ **PATHNAME**

The functions **lpntype** and **lpnpathname** extract the type and pathname of a labeled pathname, respectively.

lpntype : **LABELED_PATHNAME** $\rightarrow$ **PATHNAMETYPE**
lpnpathname : **LABELED_PATHNAME** $\rightarrow$ **PATHNAME**

$\forall$ **lpn** : **LABELED_PATHNAME**; **pnt** : **PATHNAMETYPE** $\bullet$
lpntype(**lpn**) = **pnt** $\Leftrightarrow$ ($\exists$ **pn** : **PATHNAME** $\bullet$ (**pnt**, **pn**) = **lpn**)

$\forall$ **lpn** : **LABELED_PATHNAME**; **pn** : **PATHNAME** $\bullet$
lpnpathname(**lpn**) = **pn** $\Leftrightarrow$ ($\exists$ **pnt** : **PATHNAMETYPE** $\bullet$ (**pnt**, **pn**) = **lpn**)

AbsolutePathname

Operations at the UFS level accept input of type **LABELED_PATHNAME**. The **AbsolutePathname** schema translates from a **LABELED_PATHNAME** input to the (**startfid**?, **pn**?) pair used by the **BasicAccessSystem** operations. **AbsolutePathname** is designed for piping.

AbsolutePathname

UFS
lpn? : **LABELED_PATHNAME**
pid! : **PID**
startfid! : **FID**
pn! : **PATHNAME**

---

pid! = **cpid**

startfid! = **lpntype**(**lpn**?) = **relative**
pcwd(**cpid**)
Rootfid

pn! = **lpnpathname**(**lpn**?)

Create

The **CreateOK_UFS** operation represents a successful call `Create` to the Unix kernel. We will describe error returns and error codes later.

For a **Create** operation, the process must have **r** and **x** access to the directory in which the file is to be created. **CreateUFS** takes as input the pathname for the new file (**lpn**?). **AbsolutePathname** is used to translate the labeled pathname **lpn**? to **startfid**? and **pn**?, which are the inputs expected by the **BasicAccessSystem** operations. At the UFS level **Create** combines **CreateBAS** and **OpenBAS**, so that it both creates a new file and produces a new **FD** for that file an output.

The permission checking schema, **CreatePermittedDAC**, checks that current process has the proper permissions for the directory in which the new file is to be created.

CreatePermittedDAC

DAC
BasicAccessSystem
DACVisiblePathname
startfid? : **FID**
pn? : **PATHNAME**

---

front(**pn**?) ∈ **DACvisible_pathnames**(**pid**?, **startfid**?)

(**fid** == **pathname_fid**(**startfid**?, **front**(**pn**?)) •
DAC_necessary_perms(**create**, **rdwr**) ⊆ **DAC_current_perms**(**pid**?, **fid**))

Note that it is possible to create a file with permissions that say you can't write to it. And since create opens the file, too, you get the **fd** back so you **can** write to it while it's open!

The **CreateFileAttrsDAC** operation creates the initial **DAC_File_Attributes** for a newly created file. The file's owner is taken from the power of the process creating the file. The file's group is taken from the group of its parent directory in the BSD style. (This means that we cannot create the root directory this way!) The file's permissions are passed in via the **fperms**? input.

CreateFileAttrsDAC

Δ**DAC**
BasicAccessSystem
LegalPathnames
pid? : **PID**
startfid? : **FID**
pn? : **PATHNAME**
fperms? : **FPERMS**

$$\mathbf{pathname\_fid}(\mathbf{startfid}?, \mathbf{pn}?) \notin \mathrm{dom}\,\mathbf{fdac}$$

$$\#\mathbf{pn}? > 0$$

$$(\mathbf{fid} == \mathbf{pathname\_fid}(\mathbf{startfid}?, \mathbf{pn}?) \bullet$$
$$(\mathbf{parent\_fid} == \mathbf{pathname\_fid}(\mathbf{startfid}?, \mathbf{front}(\mathbf{pn}?)) \bullet$$
$$(\mathbf{fdac}'(\mathbf{fid})).\mathbf{fperms} = \mathbf{fperms}?$$
$$\wedge\ (\mathbf{fdac}'(\mathbf{fid})).\mathbf{owner} = (\mathbf{pdac}(\mathbf{pid}?)).\mathbf{owner}$$
$$\wedge\ (\mathbf{fdac}'(\mathbf{fid})).\mathbf{group} = (\mathbf{fdac}(\mathbf{parent\_fid})).\mathbf{group}))$$

$$\mathbf{fdac} = \{\mathbf{pathname\_fid}(\mathbf{startfid}?, \mathbf{pn}?)\}\ \mathbf{fdac}'$$

$$\mathbf{pdac}' = \mathbf{pdac}$$

$$\mathbf{CreateOK\_UFS} \mathrel{\widehat=} \mathbf{AbsolutePathname}$$
$$(\mathbf{CreatePermittedDAC} \wedge \mathbf{CreateBAS} \wedge \mathbf{OpenBAS}$$
$$\wedge\ \mathbf{CreateFileAttrsDAC})$$
$$\wedge\ \Delta\mathbf{DAC}$$

Open

The **OpenOK_UFS** operation represents a successful call to `open` in the Unix kernel. We will describe error returns and error codes later.

We have not tried to model all options of the Unix `open` system call. However, we believe that the schemas defined so far are sufficient to do so.

Under Unix, **Open** may either open an existing file (as in lower levels) or create a file. We model this by having **Open** call **Create** if the named file does not exist.

OpenPermittedDAC

DAC
BasicAccessSystem
DACVisiblePathname
startfid? : **FID**
pn? : **PATHNAME**
access_mode? : **ACCESS_MODE**

$$(\mathbf{fid} == \mathbf{pathname\_fid}(\mathbf{startfid}?, \mathbf{pn}?) \bullet$$
$$\mathbf{DAC\_necessary\_perms}(\mathbf{open}, \mathbf{access\_mode}?) \subseteq \mathbf{DAC\_current\_perms}(\mathbf{pid}?, \mathbf{fid}))$$

$$\mathbf{OpenExistingFileOK\_UFS} \mathrel{\widehat=} \mathbf{AbsolutePathname}(\mathbf{OpenPermittedDAC} \land \mathbf{OpenBAS}) \land \Xi\mathbf{DAC}$$

Link and Unlink

The operations **LinkOK_UFS** describes the outcome of a successful link operation on a **UFS** state. It takes two arguments of type **LABELED_PATHNAME**, **lpn**? and **xlpn**?. The input **lpn**? is the name of the new line to be created, and must not name an existing file. The input **xlpn**? must name an existing, regular file. Both **xlpn**? and **front**(**lpn**?) must be pathnames visible to the current process.

LinkPermittedDAC

$$\begin{array}{l}
\mathbf{DAC} \\
\mathbf{BasicAccessSystem} \\
\mathbf{DACVisiblePathnames} \\
\mathbf{pid?} : \mathbf{PID} \\
\mathbf{startfid?}, \mathbf{xstartfid?} : \mathbf{FID} \\
\mathbf{pn?}, \mathbf{xpn?} : \mathbf{PATHNAME} \\
\hline
\mathbf{front}(\mathbf{pn?}) \in \mathbf{DACvisible_pathnames}(\mathbf{pid?}, \mathbf{startfid?}) \\
\mathbf{xpn?} \in \mathbf{DACvisible_pathnames}(\mathbf{pid?}, \mathbf{xstartfid?}) \\
(\mathbf{fid} == \mathbf{pathname_fid}(\mathbf{startfid?}, \mathbf{front}(\mathbf{pn?})) \bullet \\
\quad \mathbf{DAC_necessary_perms}(\mathbf{link}, \mathbf{rdwr}) \subseteq \mathbf{DAC_current_perms}(\mathbf{pid?}, \mathbf{fid}))
\end{array}$$

$$\begin{array}{l}
\mathbf{LinkOK_UFS} \mathrel{\widehat=} (\mathbf{AbsolutePathname} \land \\
\quad \mathbf{AbsolutePathname}[\mathbf{xlpn?}/\mathbf{lpn?}, \mathbf{xstartfid!}/\mathbf{startfid!}, \mathbf{xpn!}/\mathbf{pn!}]) \\
\quad (\mathbf{LinkPermittedDAC} \land \\
\quad\quad \mathbf{LinkBAS}) \\
\quad \land \Xi\mathbf{DAC}
\end{array}$$

UnlinkPermittedDAC

$$\begin{array}{l}
\mathbf{DAC} \\
\mathbf{BasicAccessSystem} \\
\mathbf{LegalPathnames} \\
\mathbf{DACVisiblePathname} \\
\mathbf{pid?} : \mathbf{PID} \\
\mathbf{startfid?} : \mathbf{FID} \\
\mathbf{pn?} : \mathbf{PATHNAME} \\
\hline
\mathbf{fid} == \mathbf{pathname_fid}(\mathbf{startfid?}, \mathbf{pn?}) \bullet \\
\quad \mathbf{DAC_necessary_perms}(\mathbf{unlink}, \mathbf{rdwr}) \subseteq \mathbf{DAC_current_perms}(\mathbf{pid?}, \mathbf{fid})
\end{array}$$

UnlinkOK_UFS $\widehat{=}$ **AbsolutePathname**
(**UnlinkPermittedDAC** $\wedge$ **RegularFile** $\wedge$ **UnlinkBAS**)
$\wedge$ Δ**DAC**

The UFS requirement that

dom **fdac** = dom **fcontents**

implies that if **UnlinkOK_UFS** results in the file being GC'd, then the DAC file attributes (**fdac**(**fid**)) must be "GC'd" as well. **DestroyFileAttrDAC** describes removing **fid** from the domain of **fdac**, but **DestroyFileAttrDAC** is never invoked explicitly.

Read, Write, Close

Read and write operations are permitted on open files. The intended DAC constraint is that a process can only read or write files to which it has permission. But the **open** operation/kernel call acts as a "gatekeeper" for read, write, and close. So once the **open** operation has checked permission, read and write need only check that the file was opened in an appropriate access mode. The operation **WriteOK_UFS** combines the two **BasicAccessSystem** operations **WriteBAS** and **AppendBAS**, as the Unix kernel call **write** does.

If the permissions associated with a file are constant, then a process will only be permitted to read or write a file to which it has the appropriate permission. However, if file permissions can change dynamically, then the file system no longer has this property. Rather, a weaker condition holds: a process will only be permitted to read or write a file to which it or an ancestor had appropriate permission when the **Open** operation was performed.

ReadOK_UFS $\widehat{=}$ **UFS** $\wedge$ **ReadBAS**[**cpid**/**pid**?]
$\wedge$ Ξ**DAC**

CloseOK_UFS $\widehat{=}$ **UFS** $\wedge$ **CloseBAS**[**cpid**/**pid**?]
$\wedge$ Ξ**DAC**

WriteOK_UFS $\widehat{=}$ **UFS**
$\wedge$ (((**FdToOid**[**cpid**/**pid**?]**OpenedForWrite**) $\wedge$ **WriteBAS**[**cpid**/**pid**?])
$\vee$ ((**FdToOid**[**cpid**/**pid**?]**OpenedForAppend**) $\wedge$ **AppendBAS**[**cpid**/**pid**?]))
$\wedge$ Ξ**DAC**

We use the form **predicate**$_i$ $\wedge$ **state_transition**$_i$ *to indicate that the* i^{th} *state_transition only applies when the* i^{th} *predicate holds. We use this form, rather than phrasing it as* **predicate**$_i$ $\Rightarrow$ **state_transition**$_i$*, because the implication is vacuously true if the predicate is not satisfied. Thus the schema is vacuously satisfied if none of the predicates are satisfied. When we add error reporting and return codes to our model, we want to be sure that the schemas modeling successful transitions are not also satisfied (even vacuously) under conditions that should report an error.*

Remark on the Z

11 Mandatory Access Controls

We treat the specification of the MAC layer analogously to the specification of the DAC layer. Both access control layers have an access control policy that controls an access mechanism through an interface-language of "permissions". The DAC and MAC policies take consider different attributes of processes and files, and they deal with different languages of permissions, but the same abstract structure describes both.

The Synergy MAC layer uses *extended permissions* to provide finer-grained control of file access. Each file and each process has an associated *security context*, which identifies the security-related attributes needed to make a policy decision. The MAC policy produces a set of extended permissions describing permitted access based on the security contexts of the user requesting access and the file being accessed.

The MAC operations and permissions described here are based on informal discussions of the Synergy file system in Fall 1994. These specifications do not reflect the full semantics of Synergy prototype.

$$\begin{aligned}\mathbf{MAC\_PERM} ::= {} & \mathbf{read\_perm} \mid \mathbf{write\_perm} \mid \mathbf{exec\_perm} \mid \mathbf{append\_perm} \mid \\ & \mathbf{create\_perm} \mid \mathbf{delete\_perm} \mid \mathbf{search\_perm} \mid \\ & \mathbf{get\_context\_perm} \mid \mathbf{set\_context\_perm}\end{aligned}$$

$$\mathbf{MAC\_PERMS} == (\mathbf{MAC\_PERM})$$

$$[\mathbf{SECCONTEXT}]$$

As in the DAC layer, we define schemas to encapsulate the file and process attributes relevant to the policy decision.

MAC_File_Attributes

$$\mathbf{context} : \mathbf{SECCONTEXT}$$

MAC_Process_Attributes

$$\mathbf{context} : \mathbf{SECCONTEXT}$$

As in the DAC layer, we define the set of operations a process may perform on a file. The policy decision is based on the MAC attributes of the process and the file, and the operation being requested.

The SFS open operation accepts several modifiers, called modes. The description of an operation is a pair, a **MACops** and an **EXT_MODE**.

$$\begin{aligned}\mathbf{MACops} ::= {} & \mathbf{mac\_open} \mid \mathbf{mac\_link} \mid \mathbf{mac\_unlink} \mid \mathbf{mac\_create} \mid \\ & \mathbf{mac\_search} \mid \mathbf{mac\_append} \mid \mathbf{mac\_delete} \mid \\ & \mathbf{mac\_get\_context} \mid \mathbf{mac\_set\_context}\end{aligned}$$

$$\begin{aligned}\mathbf{EXT\_MODE} ::= {} & \mathbf{xm\_read} \mid \mathbf{xm\_write} \mid \mathbf{xm\_readwrite} \mid \mathbf{xm\_append} \mid \\ & \mathbf{xm\_truncate} \mid \mathbf{xm\_excl}\end{aligned}$$

We define the MAC schema quite analogously to the earlier DAC schema. The two functions **fmac** and **pmac** map files and processes to their MAC-related attributes. The function **MAC_current_perms** maps a **pid** and an **fid** to the set of extended permissions describing how that process is permitted to operate on that file. The definition of that set of permissions is encapsulated in the function **MAC_policy_perms**. This policy computation considers only the relevant MAC attributes of the process and file in question (i.e., their **SECCONTEXTs**). We provide only the signature of **MAC_policy_perms**, leaving the actual policy unspecified. Finally, **MAC_necessary_perms** produces the set of permissions required for an operation to complete without error.

MAC

$$\mathbf{fmac} : \mathbf{FID}\ \ \mathbf{MAC\_File\_Attributes}$$
$$\mathbf{pmac} : \mathbf{PID}\ \ \mathbf{MAC\_Process\_Attributes}$$
$$\mathbf{MAC\_current\_perms} : \mathbf{PID} \times \mathbf{FID}\ \ \mathbf{MAC\_PERMS}$$
$$\mathbf{MAC\_necessary\_perms} : \mathbf{MACops} \times \mathbf{EXT\_MODE}\ \ \mathbf{MAC\_PERMS}$$
$$\mathbf{MAC\_policy\_perms} : (\mathbf{SECCONTEXT} \times \mathbf{SECCONTEXT})\ \ \mathbf{MAC\_PERMS}$$

$$\forall\, \mathbf{pid} : \mathbf{PID};\ \ \mathbf{fid} : \mathbf{FID} \bullet$$
$$\mathbf{MAC\_current\_perms}(\mathbf{pid}, \mathbf{fid}) =$$
$$\mathbf{MAC\_policy\_perms}((\mathbf{pmac}\ \mathbf{pid}).\mathbf{context}, (\mathbf{fmac}\ \mathbf{fid}).\mathbf{context})$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_open}, \mathbf{xm\_read}) = \{\mathbf{read\_perm}\}$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_open}, \mathbf{xm\_readwrite}) = \{\mathbf{read\_perm}, \mathbf{write\_perm}\}$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_open}, \mathbf{xm\_write}) = \{\mathbf{write\_perm}\}$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_open}, \mathbf{xm\_append}) = \{\mathbf{append\_perm}\}$$
$$\forall\, \mathbf{ext\_mode} : \mathbf{EXT\_MODE} \bullet$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_search}, \mathbf{ext\_mode}) = \{\mathbf{search\_perm}\} \wedge$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_create}, \mathbf{ext\_mode}) = \{\mathbf{create\_perm}\} \wedge$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_delete}, \mathbf{ext\_mode}) = \{\mathbf{delete\_perm}\} \wedge$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_get\_context}, \mathbf{ext\_mode}) = \{\mathbf{get\_context\_perm}\} \wedge$$
$$\mathbf{MAC\_necessary\_perms}(\mathbf{mac\_set\_context}, \mathbf{ext\_mode}) = \{\mathbf{set\_context\_perm}\}$$

We could have used a different decomposition for these security-related system components. For example, we might define one schema for **fmac** and **pmac**, which represent file and process attributes that normally change during system operation, and a separate schema for **MAC_policy_perms** and **MAC_necessary_perms**, which generally change less often, if at all, during system operation. That way we distinguish the policy and enforcement tests from dynamic process and file attributes. So then even operations that may dynamically alter file or process attributes clearly leave the policy unchanged.

Analogously to the DAC layer, we define the notion of a pathname being visible to a process. We do this with two schemas. **MACVisiblePathnames** introduces the function **MAC_visible_pathnames**, which maps a process and a starting **fid** to the set of pathnames that process may "see" in a given file system state. The schema **MACVisiblePathname** takes inputs **pid**?, and (**startfid**?, **pn**?) and asserts the property that **pn**? is a member of the process's **Mac_visible_pathnames**.

We may want to include both DAC and MAC access controls at the SFS level. This is easily done by including the both relevant MAC schema and the relevant DAC schema in the SFS definition. For example, if we want to assure that a pathname is visible under both the MAC and DAC policies, we might include both **MACVisiblePathname** and **DACVisiblePathname** in a schema definition.

MACVisiblePathnames

FileSpace
LegalPathnames
MAC
MACvisible_pathnames : **PID** × **FID** → (**PATHNAME**)

$$\forall \mathbf{pid} : \mathbf{PID};\ \mathbf{fid} : \mathbf{FID};\ \mathbf{pn} : \mathbf{PATHNAME} \bullet$$
$$\langle\rangle \in \mathbf{MACvisible\_pathnames}(\mathbf{pid}, \mathbf{fid})$$
$$\wedge$$
$$(\#\mathbf{pn} > 0$$
$$\wedge\ \mathbf{pn} \in \mathbf{MACvisible\_pathnames}(\mathbf{pid}, \mathbf{fid})$$
$$\Leftrightarrow \mathbf{front}(\mathbf{pn}) \in \mathbf{MACvisible\_pathnames}(\mathbf{pid}, \mathbf{fid})$$
$$\wedge\ (\mathbf{fid}, \mathbf{pn}) \in \mathbf{legal\_pathnames}$$
$$\wedge\ \mathbf{MAC\_necessary\_perms}(\mathbf{mac\_search}, \mathbf{xm\_read})$$
$$\subseteq \mathbf{MAC\_current\_perms}(\mathbf{pid}, \mathbf{pathname\_fid}(\mathbf{fid}, \mathbf{front}(\mathbf{pn}))))$$

MACVisiblePathname

MACVisiblePathnames
pid? : **PID**
startfid? : **FID**
pn? : **PATHNAME**

$$\mathbf{pn}? \in \mathbf{MACvisible\_pathnames}(\mathbf{pid}?, \mathbf{startfid}?)$$

12 The Synergy File System Interface

This section provides the *form* for describing the SFS layer. The specific details of the operations and MAC tests are merely meant to be illustrative.

SFS

UFS
MAC

$$\mathrm{dom}\,\mathbf{fmac} = \mathrm{dom}\,\mathbf{fcontents}$$

$$\mathrm{dom}\,\mathbf{pmac} = \mathrm{dom}\,\mathbf{pfdtable}$$

Create

Note that **CreatePermittedMAC** checks the permissions of the directory in which the new file would reside.

CreatePermittedMAC

$$
\begin{array}{l}
\mathbf{MAC} \\
\mathbf{BasicAccessSystem} \\
\mathbf{MACVisiblePathname} \\
\mathbf{startfid?} : \mathbf{FID} \\
\mathbf{pn?} : \mathbf{PATHNAME} \\
\hline
\mathbf{front}(\mathbf{pn?}) \in \mathbf{MACvisible\_pathnames}(\mathbf{pid?}, \mathbf{startfid?}) \\
(\mathbf{fid} == \mathbf{pathname\_fid}(\mathbf{startfid?}, \mathbf{front}(\mathbf{pn?})) \bullet \\
\quad \mathbf{MAC\_necessary\_perms}(\mathbf{mac\_create}, \mathbf{xm\_readwrite}) \\
\qquad \subseteq \mathbf{MAC\_current\_perms}(\mathbf{pid?}, \mathbf{fid}))
\end{array}
$$

The operation **CreateFileAttrsMAC** creates the initial MAC file attributes for a new file. It will probably share with (or perhaps implement) the definition of the **SetContext** operation. However, since it is used here as a part of the **Create** operation, it need not perform any MAC checks.

CreateFileAttrsMAC

$$
\begin{array}{l}
\Delta\mathbf{MAC} \\
\mathbf{LegalPathnames} \\
\mathbf{seccontext?} : \mathbf{SECCONTEXT} \\
\mathbf{pid?} : \mathbf{PID} \\
\mathbf{startfid?} : \mathbf{FID} \\
\mathbf{pn?} : \mathbf{PATHNAME} \\
\hline
\mathbf{pathname\_fid}(\mathbf{startfid?}, \mathbf{pn?}) \notin \mathrm{dom}\, \mathbf{fmac} \\
\#\mathbf{pn?} > 0 \\
\mathbf{fmac} = \{\mathbf{pathname\_fid}(\mathbf{startfid?}, \mathbf{pn?})\}\ \mathbf{fmac}' \\
(\mathbf{fmac}'(\mathbf{pathname\_fid}(\mathbf{startfid?}, \mathbf{pn?}))).\mathbf{context} = \mathbf{seccontext?} \\
\mathbf{pmac}' = \mathbf{pmac}
\end{array}
$$

CreateFileAttrsMAC

We conjoin the predicate properties of **CreatePermittedMAC** and **CreateOK_UFS**. So the the overall **CreateOK_SFS** operation is only defined when all of their properties are satisfied (e.g., both MAC and DAC tests).

$$
\begin{array}{l}
\mathbf{CreateOK\_SFS} \mathrel{\widehat{=}} (\mathbf{AbsolutePathname} \\
\qquad (\mathbf{CreatePermittedMAC} \wedge \mathbf{CreateFileAttrsMAC})) \\
\quad \wedge\ \mathbf{CreateOK\_UFS}
\end{array}
$$

Open

OpenPermittedMAC

MAC
BasicAccessSystem
MACVisiblePathname
startfid? : **FID**
pn? : **PATHNAME**
xmode? : **EXT_MODE**

(**fid** == **pathname_fid**(**startfid**?, **pn**?) •
  MAC_necessary_perms(**mac_open**, **xmode**?)
    ⊆ **MAC_current_perms**(**pid**?, **fid**))

OpenExistingFileOK_SFS ≙ (**AbsolutePathnameOpenPermittedMAC**)
  ∧ **OpenExistingFileOK_UFS**
  ∧ Ξ**MAC**

Link

LinkPermittedMAC

MAC
BasicAccessSystem
MACVisiblePathnames
pid? : **PID**
startfid?, **xstartfid**? : **FID**
pn?, **xpn**? : **PATHNAME**

---

front(**pn**?) ∈ **MACvisible_pathnames**(**pid**?, **startfid**?)

xpn? ∈ **MACvisible_pathnames**(**pid**?, **xstartfid**?)

(**fid** == **pathname_fid**(**startfid**?, **front**(**pn**?)) •
  MAC_necessary_perms(**mac_link**, **xm_readwrite**)
    ⊆ **MAC_current_perms**(**pid**?, **fid**))

LinkOK_SFS ≙ (**AbsolutePathname** ∧
  AbsolutePathname[**xlpn**?/**lpn**?, **xstartfid**!/**startfid**!, **xpn**!/**pn**!])
    (**LinkPermittedMAC**
      ∧ **LinkOK_UFS**)
  ∧ Ξ**MAC**

Unlink

UnlinkPermittedMAC

MAC
BasicAccessSystem
LegalPathnames
MACVisiblePathname
pid? : **PID**
startfid? : **FID**
pn? : **PATHNAME**

---

fid == **pathname_fid**(**startfid**?, **front**(**pn**?)) •
  MAC_necessary_perms(**mac_unlink**, **xm_readwrite**)
    ⊆ **MAC_current_perms**(**pid**?, **fid**)

The **Unlink_SFS** operation removes an entry from the directory (if MAC and DAC permissions allow it). The MAC file attributes associated with the file are removed when the file is GC'd from the file space (see **FileSpaceGC**, **CloseBAS**, and **UnlinkBAS**). This follows from the requirement that in all **SFS** states the domain of **fmac** is the same as the domain of **fcontents**. File attributes are associated with files, not with directory entries. Hence they are not necessarily affected each time a directory entry is deleted.

UnlinkOK_SFS ≙ (**AbsolutePathname**
    (**UnlinkPermittedMAC** ∧ **UnlinkOK_UFS**))
  ∧ Δ**MAC**

Read, Write, Close

ReadOK_SFS ≙ **ReadOK_UFS**

WriteOK_SFS ≙ **WriteOK_UFS**

CloseOK_SFS ≙ **CloseOK_UFS**

Get_Context

GetContextPermittedMAC

MAC
LegalPathname
MACVisiblePathname
pid? : **PID**
startfid? : **FID**
pn? : **PATHNAME**

(**fid** == **pathname_fid**(**startfid**?, **pn**?) •
  MAC_necessary_perms(**mac_get_context**, **xm_readwrite**)
    ⊆ **MAC_current_perms**(**pid**?, **fid**))

GetContext

MAC
LegalPathname
fid? : **FID**
seccontext! : **SECCONTEXT**

---

seccontext! = (**fmac**(**fid**?)).**context**

GetContextOK_SFS $\widehat{=}$ **AbsolutePathname**
(**GetContextPermittedMAC** $\wedge$
(**PathnameLookupGetContext**))

13 Reactions to the Z notation

We are both impressed with the simplicity of simple specifications in Z, and with the complexity of anything larger.

1. Z encourages defining a system state in terms a large number of state variables. It provides little mechanism for structuring the state.

2. Standard Z usage requires that a schema representing a state transition specify the new value of all state variables.

   That is, the inclusion of $\Delta\mathbf{S}$ in a schema representing a state transition on the system state described in schema $\mathbf{S}$ introduces all of the variables in $\mathbf{S}'$ (i.e., all of the primed state-variables). If the value any of these primed variables are not specified in the state-transition schema, then those values are completely unconstrained and completely unrelated to the corresponding initial state variable (i.e., the corresponding unprimed variable). Since there tend to be many state variables, this mechanism is error prone.

3. Many simple errors in Z specifications prove difficult to uncover.

   (a) Use of redundant input/output variables in schemas.

   If you don't include redundant variable declarations in a schema that inherits them from another schema, then the very variables of interest to your schema are not made explicit in it.

   If you do include redundant variable declarations, there is no protection from misspelling variables names.

   (b) Sometimes primed variables are left unspecified in a schema, because that schema will later be combined with other schemas that do specify that variable. But there is neither a mechanism to declare this intent, nor an easy mechanism to detect which variables are left unspecified. This problem is particularly noticeable as the number state variables gets large, and you attempt to combine schemas (via schema inclusion or composition).

   Type checking is a very useful means of finding first-order errors in a Z specification. But it by no means enough.

We have not investigated the "object oriented" variations of Z. Perhaps by applying the structuring principles of object-oriented programming to Z specifications, larger-sized specifications will become more manageable.

As part of this effort an executable model of this specification was written using the ACL2 system. Rephrasing the specification in ACL2 exposed some minor errors. Getting the model accepted by the ACL2 system amounts to a proof that the model is consistent.

14 Errors Uncovered by ACL2 Model

Here is a summary of errors detected during the development of this Z specification by building an ACL2 model of the specification. The ACL2 model only went as far as the DAC layer. (These errors have been corrected, and so do not appear in the Z specification in this report.)

1. *Input name mismatch.* In the schema **LinkFS** the schema **RegularFile** was included "above the line." **RegularFile** was intended to apply to the existing file, but **RegularFile** relates to inputs **startfid**? and **pn**?, whereas the existing file input for **LinkFS** is **fid**?.

2. *Contradiction.* The schemas **RegularFile** and **NonExistentFile** are contradictory; that is, they denote disjoint sets of bindings, and so no bindings satisfy both schemas. Since **LinkFS** included them both, no bindings would have satisfied **LinkFS**. Of course, **RegularFile** was intended to apply to the existing file, and **NonExistentFile** was intended to apply to the new pathname for that file.

3. *Incorrect specification.* The schema **OpenFT** incorrectly truncated files when opening them for writing, even if the file was being opened in append mode.

4. *Use of undefined expressions.* Zed permits the use of expressions which mention a function applied to arguments not in the domain of that function. Zed says such expressions are meaningless. However, it is not always obvious that the author meant to write a meaningless expression.

   In the schema **OnlyOpenDirectoryRDONLY** the expression **ftype**(**fid**?) is used without requiring **fid**? $\in$ dom **ftype**. Thus, sometimes the schema's predicate is meaningless. We may know (or believe) that the context of use of always requires **fid**? $\in$ dom **ftype**, because most other operations require it. ACL2 requires that functions only be applied to arguments in their domains. So this casual style was not permitted.

5. *Framing error.* The **LinkBAS** operation did not specify that state variables **ftype** and **RootFid** remained unchanged.

6. *Framing error.* The **CreateBAS** operation did not specify that state variables **FileTable** and **ProcessTable** remained unchanged. However, the real error was that **CreateBAS** should not have been Δ**BasicAccessSystem** at all!

7. *Importing error.* The schema **LinkFS** imported the schema **LegalPathname**, rather than the schema **LegalPathnames**. The former specifies that the pathname identified by (**startfid**?, **pn**?) names an existing file. The schema **LegalPathnames** just introduces the **legal_pathnames** (the set of legal pathnames) as a schema variable. The properties "below the line" in **LinkFS** then related its inputs to the set **legal_pathnames**.

References

[Bach, 1986] Maurice J. Bach.
The Design of the Unix Operating System.
Prentice-Hall, Englewood Cliffs, New Jersey, 07632, 1986.

[CSRG, 1986] CSRG.
Unix User's Reference Manual.
Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, California, 4.3 berkeley softwar distribution edition, 1986.

[Leffler *et al.*, 1989] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman.
The Design and Implementation of the 4.3BSD Unix Operating System.
Addison-Wesley, 1989.

[Morgan and Sufrin, 1987] Carol Morgan and Bernard Sufrin.
Specification for the unix filing system.
In Ian Hayes, editor, *Specification Case Studies*, pages 91–140. Prentice-Hall, 1987.

[Spivey, 1989] J.M. Spivey.
The Z Notation: A Reference Manual.
Prentice Hall, 1989.

[Stevens, 1992] W. Richard Stevens.
Advanced Programming in the UNIX Environment.
Addison-Wesley, 1992.

Index