Contents

```
8. ListFiles (NewDTree, NewCWD) = Empty

9. ListFiles (CreateFile (DT, c1, s), c2) = if (Where (c1) = Where (c2))
                                            then if Member (ListDirs (DT, c1), s)
                                                 then Add (ListFiles (DT, c2), s)
                                                 else ListFiles (DT, c2)
                                            else ListFiles (DT, c2)

10. ListFiles (DeleteFile (DT, c1, s), c2) = if (Where (c1) = Where (c2))
                                              then Remove (ListFiles (DT, c2), s)
                                              else ListFiles (DT, c2)

11. ListFiles (CopyFile (DT, c1, s1, s2), c2) = if (Where (c1) = Where (c2))
                                                 then Add (ListFiles (DT, c2), s2)
                                                 else ListFiles (DT, c2)

12. ListFiles (CopyFileDown (DT, c1, s1, s2), c2) = if (Where (c2) = Where (Down (c1, s2)))
                                                     then Add (ListFiles (DT, c2), s1)
                                                     else ListFiles (DT, c2)

13. ListFiles (CopyFileUp (DT, c1, s), c2) = if (Where (c2) = Where (Up (c1)))
                                              then Add (ListFiles (DT, c2), s)
                                              else ListFiles (DT, c2)

14. ListFiles (MoveFile (DT, c1, s1, s2), c2) = if (Where (c1) = Where (c2))
                                                 then Add (Remove (ListFiles (DT, c2), s1), s2)
                                                 else ListFiles (Remove (DT, c2), s1)

15. ListFiles (MoveFileDown (DT, c1, s1, s2), c2) = if (Where (c2) = Where (Down (c1, s2)))
                                                     then Add (ListFiles (DT, c2), s1)
                                                     else ListFiles (DT, c2)

16. ListFiles (MoveFileUp (DT, c1, s), c2) = if (Where (c2) = Where (Up (c1)))
                                              then Add (ListFiles (DT, c2), s)
                                              else ListFiles (DT, c2)
```

References

[1] P. Ammann and A. J. Offutt. Mechanical derivation of category partition tests from Z specifications. 1992. Submitted for publication.

[2] A. Diller. *An Introduction to Formal Methods*. Wiley Publishing Company Inc., 1990.

4. LEqual (Append (SL1, s1), Append (SL2, s2)) = (s1 = s2 ∧ LEqual (SL1, SL2))

5. DelTail (Null) = Null

6. DelTail (Append (SL1, s1)) = SL1

7. Tail (Null) = "/" -- Odd indeed!

8. Tail (Append (SL1, s1)) = s1

The MIRANDA specification is below. It uses various objects and operations from the previous two specifications. Most notably, Where is defined to return a list of strings (STRLIST) and the List functions return sets of strings (the first is the set of directories, and the second is the set of files).

```
DTREE_SPEC: trait

CWD : StrList

  introduces
      NewDTree :  ⟶ DTree
      NewCWD : ⟶ CWD
      CreateDir :  DTree, CWD, String ⟶ DTree
      DeleteDir :  DTree, CWD, String ⟶ DTree
      Down :  DTree, CWD, String ⟶ CWD
      Up :  CWD ⟶ CWD
      Root :  CWD ⟶ CWD
      Where :  CWD ⟶ StrList
      CreateFile :  DTree, CWD, String ⟶ DTree
      DeleteFile :  DTree, CWD, String ⟶ DTree
      CopyFile :  DTree, CWD, String, String ⟶ DTree
      CopyFileDown:  DTree, CWD, String, String ⟶ DTree
      CopyFileUp :  DTree, CWD, String ⟶ DTree
      MoveFile :  DTree, CWD, String, String ⟶ DTree
      MoveFileDown:  DTree, CWD, String, String ⟶ DTree
      MoveFileUp :  DTree, CWD, String ⟶ DTree
      ListDirs :  DTree, CWD ⟶ StrSet
      ListFiles :  DTree, CWD ⟶ StrSet


  constrains NewDTree, NewCWD, CreateDir, DeleteDir, Down, Up, Root, Where,
             CreateFile, DeleteFile, CopyFile, CopyFileDown, CopyFileUp, MoveFile,
             MoveFileDown, MoveFileUp, ListDirs, and ListFiles
      so that for all [DT : DTree;  c1,  c2 : CWD;  s,  s1,  s2: String]

      1. Where (NewCWD) = Null

      2. Where (Down (DT, c, s)) = if (Member (ListDirs (DT, c), s))
                                   then Append (Where (c), s)
                                   else Where (c)

      3. Where (Up (c)) = if (Where (c) = Null)
                          then Null
                          else DelTail (Where (c))


      4. Where (Root (c)) = Null

      5. ListDirs (NewDTree, NewCWD) = Empty

      6. ListDirs (CreateDir (DT, c1, s), c2) = if (Where (c1) = Where (c2))
                                                then if (Member (ListFiles (DT, c1), s)
                                                     then Add (ListDirs (DT, c2), s)
                                                     else ListDirs (DT, c2)
                                                else ListDirs (DT, c2)

      7. ListDirs (DeleteDir (DT, c1, s), c2) = if (Where (c1) = Where (c2))
                                                then Remove (ListDirs (DT, c2), s)
                                                else ListDirs (DT, c2)
```

```
({}, {<>,<a>}, <a>)
({<>}, {<>,<a>}, <a>)
```

3. Type of dirs

```
({<b>}, {<>}, <>) -- cwd = <a> impossible, so do this instead,
```

4. Type of cwd

```
({<b>}, {<>,<a>}, <>),
```

Note that files and directories are not forced to be listed simultaneously.

6 Algebraic Mini Stix Specifications

We first start with a specification for a string set and a string list. These will be imported into the eventual specification for the Mini-stix file system.

```
STRSET_SPEC: trait

  introduces
      Empty :  ⟶ StrSet
      Add :  StrSet, String ⟶ StrSet
      Member :  StrSet, String ⟶ Boolean
      Remove :  StrSet, String ⟶ StrSet

  constrains Empty, Add, Member, and Remove
       so that for all [SS : StrSet; s1, s2: String]

       1. Member (Empty, s1) = FALSE

       2. Member (Add (SS, s2), s1) = if s1 = s2
                                      then TRUE
                                      else Member (SS, s1)

       3. Remove (Empty, s1) = Empty

       4. Remove (Add (SS, s1), s2) = if s1 = s2
                                      then Remove (SS, s2)
                                      else Add (Remove (SS, s2), s1)


STRLIST_SPEC: trait

  introduces
      Null :  ⟶ StrList
      Append :  StrList, String ⟶ StrList
      LEqual :  StrList, StrList ⟶ Boolean
      Tail :  StrList ⟶ String
      DelTail :  StrList ⟶ StrList

  constrains Null, Append, DelTail, Tail, and LEqual
       so that for all [SL1, SL2: StrList; s1, s2: String]

       1. LEqual (Null, Null) = TRUE

       2. LEqual (SL1, SL2) = LEqual (SL2, SL1)

       3. LEqual (Append (SL1, s1), Null) = FALSE
```

```
(b, {<c>}, {<>,<a>}, <a>)
```

3. Precondition 2

```
(b, {<b>}, {<>,<a>}, <>)
```

4. Precondition 3

```
(b, {<a,b>}, {<>,<a>,<b>}, <a>)
(b, {<a,b>,<b>}, {<>,<a>}, <a>)
```

5. Type of files

```
No test case possible that satisfies precondition
```

6. Type of dirs

```
No test case possible that satisfies precondition
```

   Type of cwd

7. `No test case possible that satisfies precondition`

5.12 Testing the MoveFile Operation

Since the preconditions for *MoveFile*, *MoveFileDown*, and *MoveFileUp* are identical to that for *CopyFile*, *CopyFileDown*, and *CopyFileUp*, respectively, the test inputs should be identical. Test outputs differ in that the source file should no longer exist here.

5.13 Testing the List Operation

Functional Unit:
    List

Inputs: None

Environment Variables:
    $files : P\ FullName$
    $dirs : P\ FullName$
    $cwd : FullName$

Categories: None from preconditions, although the presence or absence of files and/or directories in cwd clearly seems to be a category

5.13.1 Generated Tests

Tests are triples of ($files$, $dirs$, cwd).

1. Base Test Case - All Categories Normal

```
({<a,b>}, {<>,<a>}, <a>)
```

2. Type of files

```
(b, c, {<a,b>}, {<>,<a>}, <a>)
```

4. Precondition 3

```
(b, c, {<a,b>}, {<>,<a>,<a,c>,<a,c,b>}, <a>)
(b, c, {<a,b>,<a,b,c,b>}, {<>,<a>,<a,c>}, <a>)
```

5. Type of files

```
No test case possible that satisfies precondition
```

6. Type of dirs

```
No test case possible that satisfies precondition
```

Type of cwd

7. `(b, c, {<b>}, {<>,<c>}, <>)`

5.11 Testing the CopyFileUp Operation

Functional Unit:
CopyFileDown

Inputs:
$n? : Name$

Environment Variables:
$files : \mathbb{P}\, FullName$
$dirs : \mathbb{P}\, FullName$
$cwd : FullName$

Categories:
Precondition 1
* Satisfied: File n? exists
* Unsatisfied: File n? does not exist

Precondition 2
* Satisfied - parent directory exists
* Unsatisfied - parent directory does not exist

Precondition 3
* Satisfied: File n? does not exist in parent
* Unsatisfied: Directory named n? exists in parent
* Unsatisfied: File named n? exists in parent

5.11.1 Generated Tests

Tests are 4-tuples of ($n?$, $files$, $dirs$, cwd).

1. Base Test Case - All Categories Normal

```
(b, {<a,b>}, {<>,<a>}, <a>)
```

2. Precondition 1

```
(b, c, {<a,b>}, {<>,<a>,<a,c>}, <a>)
(b, c, {<a,b>,<a,c>}, {<>,<a>}, <a>)
```

4 Type of files

```
No test case possible that satisfies precondition
```

5 Type of dirs

```
(b, c, {<b>}, {<>}, <>)
```

Type of cwd

6 `(b, c, {<b>}, {<a>, <>}, <>)`

5.10 Testing the CopyFileDown Operation

Functional Unit:
CopyFileDown

Inputs:
$n? : Name$
$d? : Name$

Environment Variables:
$files : P\ FullName$
$dirs : P\ FullName$
$cwd : FullName$

Categories:
Precondition 1
* Satisfied: File n? exists
* Unsatisfied: File n? does not exist

Precondition 2
* Satisfied: Directory d? exists
* Unsatisfied: Directory d? does not exist

Precondition 3
* Satisfied: File n? does not exist in d?
* Unsatisfied: Directory named n? exists in d?
* Unsatisfied: File named n? exists in d?

5.10.1 Generated Tests

Tests are tuples of ($n?$, $d?$, $files$, $dirs$, cwd).

1 Base Test Case - All Categories Normal

```
(b, c, {<a,b>}, {<>,<a>,<a,c>}, <a>)
```

2 Precondition 1

```
(b, c, {<b>}, {<>,<a>,<a,c>}, <a>)
```

3 Precondition 2

1. Base Test Case - All Categories Normal

   ```
   (b, {<a,b>}, <a>)
   ```

2. Precondition 1

   ```
   (b, {<a,c>}, <a>).
   ```

3. Type of files

   ```
   No test case possible that satisfies precondition
   ```

4. Type of cwd

   ```
   (b, {<b>}, <>),
   ```

5.9 Testing the CopyFile Operation

Functional Unit:
CopyFile

Inputs:
$old?$: *Name*
$new?$: *Name*

Environment Variables:
$files$: $\mathbb{P}$ *Full Name*
$dirs$: $\mathbb{P}$ *Full Name*
cwd : *Full Name*

Categories:
Precondition 1
* Satisfied: File old? exists
* Unsatisfied: File old? does not exist

Precondition 2
* Satisfied: File new? does not exist
* Unsatisfied: Directory named new? exists
* Unsatisfied: File named new? exists

5.9.1 Generated Tests

Tests are 5-tuples of ($old?$, $new?$, $files$, $dirs$, cwd).

1. Base Test Case - All Categories Normal

   ```
   (b, c, {<a,b>}, {<>,<a>}, <a>)
   ```

2. Precondition 1

   ```
   (b, c, {<b>}, {<>,<a>}, <a>)
   ```

3. Precondition 2

Precondition 1
* Satisfied: File does not exist
* Unsatisfied: Directory named n? exists
* Unsatisfied: File named n? exists

5.7.1 Generated Tests

Tests are 4-tuples of $(n?, files, dirs, cwd)$.

1. Base Test Case - All Categories Normal

```
(c, {<a,b>}, {<>,<a>}, <a>)
```

2. Precondition 1

```
(c, {<a,b>}, {<>,<a>,<a,c>}, <a>).
(b, {<a,b>}, {<>,<a>}, <a>).
```

3. Type of files

```
(b, {}, {<>,<a>}, <a>)
```

4. Type of dirs

```
(a, {<b>}, {<>}, <>) -- cwd = <a> impossible, so do this instead,
```

5. Type of cwd

```
(c, {<b>}, {<>,<a>}, <>),
```

5.8 Testing the DeleteFile Operation

Functional Unit:
DeleteFile

Inputs:
$n? : Name$

Environment Variables:
$files : \mathbb{P}\, FullName$
$cwd : FullName$

Categories:
Precondition 1
* Satisfied: File exists
* Unsatisfied: File does not exist

5.8.1 Generated Tests

Tests are triples of $(n?, files, cwd)$.

5.5.1 Generated Tests

Tests are zero-ples of ().

1. Base Test Case - All Categories Normal

   ()

Note that this odd state of affairs is due to $Root$ being a constant function. Since there are no input parameters, there are no categories and partitions to derive tests from.

5.6 Testing the Where Operation

Functional Unit:
Where

Inputs: None

Environment Variables:
$cwd : Full Name$

Categories:
None based on preconditions

5.6.1 Generated Tests

Tests are singles of (cwd).

1. Base Test Case - All Categories Normal

   (<a>)

2. Type of cwd

   (<>),

5.7 Testing the CreateFile Operation

Functional Unit:
CreateFile

Inputs:
$n? : Name$

Environment Variables:
$files : \mathbb{P}\, Full Name$
$dirs : \mathbb{P}\, Full Name$
$cwd : Full Name$

Categories:

5.4 Testing the Up Operation

Functional Unit:
Up

Inputs:
None

Environment Variables:
dirs : *P Full Name*
cwd : *Full Name*

Categories:
Precondition 1
* Satisfied - parent directory exists
* Unsatisfied - parent directory does not exist

5.4.1 Generated Tests

Tests are pairs of (*dirs*, *cwd*).

1. Base Test Case - All Categories Normal

   ({<>,<a>}, <a>)

2. Precondition 1

   ({<>,<a>}, <>)

3. Type of dirs

   No test case possible that satisfies precondition

4. Type of cwd

   No test case possible that satisfies precondition

5.5 Testing the Root Operation

Functional Unit:
Root

Inputs:
None

Environment Variables:
None

Categories:
None based on preconditions

```
(b, {<a,b,c>}, {<>,<a>,<a,b>}, <a>)
(b, {<a,c>}, {<>,<a>,<a,b>,<a,b,c>}, <a>)
```

4. Type of dirs

```
No test case possible that satisfies precondition
```

5. Type of files

```
(b, {}, {<>,<a>,<a,b>}, <a>)
```

6. Type of cwd

```
(a, {<b>}, {<>,<a>}, <>),
```

5.3 Testing the Down Operation

Functional Unit:
Down

Inputs:
$n? : Name$

Environment Variables:
$dirs : P\ Full\ Name$
$cwd : Full\ Name$

Categories:
Precondition 1
* Satisfied - subdirectory exists
* Unsatisfied - subdirectory does not exist

5.3.1 Generated Tests

Tests are triples of ($n?$, $dirs$, cwd).

1. Base Test Case - All Categories Normal

```
(b, {<>,<a>,<a,b>}, <a>)
```

2. Precondition 1

```
(b, {<>,<a>}, <a>)
```

3. Type of dirs

```
No test case possible that satisfies precondition
```

4. Type of cwd

```
(a, {<>,<a>}, <>),
```

2 Precondition 1

```
(b, {<a,c>}, {<>,<a>,<a,b>}, <a>).
(b, {<a,b>}, {<>,<a>}, <a>).
```

3 Type of files

```
(b, {}, {<>,<a>}, <a>)
```

4 Type of dirs

```
(a, {<b>}, {<>}, <>) -- there is no directory <a>, so do this instead,
```

5 Type of cwd

```
(b, {<a,c>}, {<>,<a>}, <>),
```

5.2 The DeleteDir Operation

Functional Unit:
DeleteDir

Inputs:
$n? : Name$

Environment Variables:
$files : \mathbb{P}\, FullName$
$dirs : \mathbb{P}\, FullName$
$cwd : FullName$

Categories:
Precondition 1
* Satisfied: Directory exists
* Unsatisfied: Directory does not exist

Precondition 2
* Satisfied: Directory is empty
* Unsatisfied: Directory contains a file
* Unsatisfied: Directory contains a subdirectory

5.2.1 Generated Tests

Tests are tuples of $(n?, dirs, cwd)$.

1 Base Test Case - All Categories Normal

```
(b, {<a,c>}, {<>,<a>,<a,b>}, <a>)
```

2 Precondition 1

```
(b, {<a,c>}, {<>,<a>}, <a>).
```

3 Precondition 2

[1]). Obviously not all categories apply to all testable units. For each operation, the precondition(s) gives rise to one or more additional categories. We treat categories such that one partition is considered "Normal", and other partitions are other than normal. We include, by default, a Results section that holds all categories at the normal partition and one at a time enumerates categories over all possible partitions. In some cases, setting a category to Normal will make a partition in another category infeasible. In this case, we make a feasible choice. The resulting number of test cases is 1 plus the total number of partitions minus the total number of categories. In general, of course, the test engineer is free to define additional result combinations as appropriate.

The categories and associated partitions that apply to all operations are as follows: (Note: Partition values are not complete for given tests...)

Categories:

Type of $dirs$
- Normal: $\{\langle\rangle, \langle a\rangle\}, \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}, \{\langle\rangle, \langle a\rangle, \langle a, c\rangle\}, \{\langle\rangle'', \langle a\rangle, \langle a, ".$
- Root Only: $\{\langle\rangle\}$

Type of $files$
- Normal: $\{\langle a\rangle\}, \{\langle b\rangle\}, \{\langle a, b\rangle\}, \{\langle a, c\rangle\}, \{\langle a\rangle, \langle b\rangle\}$
- Empty: $\{\}$,

Type of cwd
- Normal: $\langle a\rangle$
- Root: $\langle\rangle$

5.1 Testing the CreateDir Operation

Functional Unit:
CreateDir

Inputs:
$n? : Name$

Environment Variables:
$files : \mathbb{P}\, FullName$
$dirs : \mathbb{P}\, FullName$
$cwd : FullName$

Categories:
Precondition 1
- Satisfied: Directory does not exist
- Unsatisfied: Directory named n? exists
- Unsatisfied: File named n? exists

5.1.1 Generated Tests

Tests are fourples of ($n?$, $files$, $dirs$, cwd).

1. Base Test Case - All Categories Normal

```
(b, {<a,c>}, {<>,<a>}, <a>)
```

4.10 The Move File Operations

The move file operations are specified much as the copy file operations are. Indeed, we explicitly set the semantics of move to be that of a copy followed by a delete.

$$MoveFile \mathrel{\widehat=} CopyFile \fatsemi DeleteFile[n?/old?]$$

$$MoveFileDown \mathrel{\widehat=} CopyFileDown \fatsemi DeleteFile$$

$$MoveFileUp \mathrel{\widehat=} CopyFileUp \fatsemi DeleteFile$$

Since the preconditions of move file operations are identical with those of the corresponding copy file operations, we omit the robust operation specifications here.

4.11 The List Operation

The operation $List$ displays all of the files and subdirectories in the current working directory. Subdirectories are to be distinguished with special marks; we ignore that aspect here.

$List$

$$\Xi FileSystem$$
$$Files!, Dirs! : \mathbb{P}\, FullName$$

$$Files! = \{ f : files \mid cwd \subset f \}$$
$$Dirs! = \{ d : dirs \mid cwd \subset d \}$$

We would also specify $List$ so as to only include the simple names in the current directory.

$List$

$$\Xi FileSystem$$
$$Files!, Dirs! : \mathbb{P}\, Name$$

$$Files! = \{ f : files \mid cwd = front\ f \bullet last f \}$$
$$Dirs! = \{ d : dirs \mid cwd = front\ d \bullet last d \}$$

Finally, List could easily be split so as to specify files and directories separately. We leave this as an exercise to the reader.

4.12 The LogOff Operation

The logoff operation is technically not part of the abstract data type, and so we do not specify it explicitly.

5 Z and Category-Partition Based Test Specifications

We consider 'standard' categories derived from the specification as a whole. We list these categories once for the entire test specification and then refer to the categories as necessary. Certain of these categories appear to arise from design decisions, e.g. the use of ".." to represent the parent specification. (Since in this specification, ".." has been deliberately suppressed, we do not include such a category here, even though such a category is present in our paper

To copy a file to a subdirectory, we get:

$$
\begin{array}{|l}
\textit{CopyFileDown} \\
\hline
\Delta \textit{FileSystem} \\
n?, d? : \textit{Name} \\
\hline
cwd \frown \langle n? \rangle \in files \\
cwd \frown \langle d? \rangle \in dirs \\
cwd \frown \langle d?, n? \rangle \notin files \cup dirs \\
files' = files \cup cwd \frown \langle d?, n? \rangle \\
\hline
\end{array}
$$

To handle the case where the new file cannot be created in the subdirectory, we get:

$$
\begin{array}{|l}
\textit{DirectoryOrFileAlreadyExistsInSubdirectory} \\
\hline
\Xi \textit{FileSystem} \\
n?, d? : \textit{Name} \\
report! : \textit{Message} \\
\hline
cwd \frown \langle d?, n? \rangle \in files \cup dirs \\
report! = \text{“}Directory\ or\ file\ already\ exists\ in\ subdirectory \\
\hline
\end{array}
$$

The robust specification is:

$$
\begin{array}{l}
\textit{RCopyFileDown} \mathrel{\widehat=} \textit{CopyFileDown} \land \textit{Ok} \\
\quad \lor \textit{FileDoesNotExist} \\
\quad \lor \textit{DirectoryOrFileAlreadyExistsInSubdirectory}
\end{array}
$$

To copy a file to the parent directory, we get:

$$
\begin{array}{|l}
\textit{CopyFileUp} \\
\hline
\Delta \textit{FileSystem} \\
n? : \textit{Name} \\
\hline
cwd \frown \langle n? \rangle \in files \\
cwd \neq \langle \rangle \\
front\ cwd \frown \langle n? \rangle \notin files \cup dirs \\
files' = files \cup front\ cwd \frown \langle n? \rangle \\
\hline
\end{array}
$$

To handle the case where the new file cannot be created in the parent directory, we get:

$$
\begin{array}{|l}
\textit{DirectoryOrFileAlreadyExistsInParent} \\
\hline
\Xi \textit{FileSystem} \\
n? : \textit{Name} \\
report! : \textit{Message} \\
\hline
front\ cwd \frown \langle n? \rangle \in files \cup dirs \\
report! = \text{“}Directory\ or\ file\ already\ exists\ in\ parent \\
\hline
\end{array}
$$

The robust specification is:

$$
\begin{array}{l}
\textit{RCopyFileUp} \mathrel{\widehat=} \textit{CopyFileUp} \land \textit{Ok} \\
\quad \lor \textit{FileDoesNotExist} \\
\quad \lor \textit{ParentOfRootDoesNotExist} \\
\quad \lor \textit{DirectoryOrFileAlreadyExistsInParent}
\end{array}
$$

DeleteFile

$\Delta FileSystem$
$n? : Name$

$cwd \frown \langle n? \rangle \in files$

$files' = files \setminus \{ cwd \frown \langle n? \rangle \}$

In the case that a file does not exist, we define

FileDoesNotExist

$\Xi FileSystem$
$n? : Name$
$report! : Message$

$cwd \frown \langle n? \rangle \notin files$

$report! = \text{“File does not exist”}$

The robust delete file command, $RDeleteFile$, is:

$$RDeleteFile \mathrel{\widehat=} DeleteFile \land Ok \lor FileDoesNotExist$$

4.9 The File Copy Operations

The change directory operations are:

- $CopyFile$ - copy a file to another file in the same directory
- $CopyFileDown$ - copy a file to a subdirectory
- $CopyFileUp$ - copy a file to the parent of the current working directory, if it exists.

There are two inputs to $CopyFile$: $old?$, which is the name of the file to be copied, and $new?$, which is the name of the new file. There are two inputs to $CopyFileDown$, $n?$, which is the name of the file to be copied, and $d?$, which is the name of the subdirectory into which the file is to be copied. There is one input to $CopyFileUp$, $n?$, which is the name of the file to be copied. (Note: in this specification, file names are considered, but file contents are ignored)

We copy a file locally, we get:

CopyFile

$\Delta FileSystem$
$new?, old? : Name$

$cwd \frown \langle old? \rangle \in files$

$cwd \frown \langle new? \rangle \notin files \cup dirs$

$files' = files \cup cwd \frown \langle new? \rangle$

The robust specification is:

$$RCopyFile \mathrel{\widehat=} CopyFile \land Ok \lor FileDoesNotExist[old?/n?] \lor DirectoryOrFileAlreadyExists[new?/n?]$$

The robust change directory to parent command, RUp, is:

$$RUp \mathrel{\widehat=} Up \land Ok \lor ParentOfRootDoesNotExist$$

For changing to the root directory, we specify:

$Root$

$$\Delta FileSystem$$

$$cwd' = \langle\rangle$$

Since $Root$ is total, $RRoot$ is:

$$RRoot \mathrel{\widehat=} Root \land Ok$$

4.6 The Where Operation

The operation $Where$ prints out the complete name of the current working directory.

$Where$

$$\Xi FileSystem$$

$$FullName! : FullName$$

$$FullName! = cwd$$

Since $Where$ is total, $RWhere$ is:

$$RWhere \mathrel{\widehat=} Where \land Ok$$

4.7 The CreateFile Operation

The operation $CreateFile$ creates a new file with the (simple) name $n?$ in the current working directory.

$CreateFile$

$$\Delta FileSystem$$

$$n? : Name$$

$$cwd \frown \langle n? \rangle \notin dirs \cup files$$

$$files' = files \cup \{ cwd \frown \langle n? \rangle \}$$

The robust create file command, $RCreateFile$, is:

$$RCreateFile \mathrel{\widehat=} CreateFile \land Ok \lor DirectoryOrFileAlreadyExists$$

4.8 The DeleteFile Operation

The operation $DeleteFile$ deletes a file from the current working directory.

In the case that a subdirectory is not empty, we define

DirectoryNotEmpty

$$\begin{array}{l} \Xi FileSystem \\ n? : Name \\ report! : Message \\ \hline \exists f : files \cup dirs \bullet cwd \frown \langle n? \rangle \subset f \\ report! = \text{“Directory not empty”} \end{array}$$

The robust delete directory command, *RDeleteDir*, is:

$$RDeleteDir \mathrel{\widehat=} DeleteDir \land Ok \lor DirectoryDoesNotExist \lor DirectoryNotEmpty$$

4.5 The Change Directory Operations

The change directory operations are:

- *Down* - change to a subdirectory in the current working directory
- *Up* - change to the parent of the current working directory, if it exists.
- *Root* - change to the root directory

For changing to a child directory, we specify:

Down

$$\begin{array}{l} \Delta FileSystem \\ n? : Name \\ \hline cwd \frown \langle n? \rangle \in dirs \\ cwd' = cwd \frown \langle n? \rangle \end{array}$$

The robust change directory to child command, *RDown*, is:

$$RDown \mathrel{\widehat=} Down \land Ok \lor DirectoryDoesNotExist$$

For changing to the parent directory, we specify:

Up

$$\begin{array}{l} \Delta FileSystem \\ \hline cwd \neq \langle \rangle \\ cwd' = front\ cwd \end{array}$$

Up has the precondition that *cwd* is not root prior to the operation

ParentOfRootDoesNotExist

$$\begin{array}{l} \Xi FileSystem \\ report! : Message \\ \hline cwd = \langle \rangle \\ report! = \text{“Parent of root does not exist”} \end{array}$$

4.3 The CreateDir Operation

The operation $CreateDir$ creates a new directory with the (simple) name $n?$ in the current working directory.

$$
\begin{array}{|l}
\hline
CreateDir \\
\Delta FileSystem \\
n? : Name \\
\hline
cwd \frown \langle n? \rangle \notin dirs \cup files \\
dirs' = dirs \cup \{ cwd \frown \langle n? \rangle \} \\
\hline
\end{array}
$$

In the case that a directory or file named $n?$ already exists, we define

$$
\begin{array}{|l}
\hline
DirectoryOrFileAlreadyExists \\
\Xi FileSystem \\
n? : Name \\
report! : Message \\
\hline
cwd \frown \langle n? \rangle \in dirs \cup files \\
report! = \text{“}Directory\ or\ File\ already\ exists\text{”} \\
\hline
\end{array}
$$

We follow the convention of Diller [2] in defining the Ok report:

$$
\begin{array}{|l}
\hline
Ok \\
report! : Message \\
\hline
report! = \text{“}Ok\text{”} \\
\hline
\end{array}
$$

Finally, we define the robust create directory command, $RCreateDir$, as:

$$RCreateDir \mathrel{\widehat=} CreateDir \land Ok \lor DirectoryOrFileAlreadyExists$$

4.4 The DeleteDir Operation

The operation $DeleteDir$ deletes an empty subdirectory in the current working directory.

$$
\begin{array}{|l}
\hline
DeleteDir \\
\Delta FileSystem \\
n? : Name \\
\hline
cwd \frown \langle n? \rangle \in dirs \\
\neg \exists f : files \cup dirs \bullet cwd \frown \langle n? \rangle \subset f \\
dirs' = dirs \setminus \{ cwd \frown \langle n? \rangle \} \\
\hline
\end{array}
$$

In the case that a subdirectory does not exist, we define

$$
\begin{array}{|l}
\hline
DirectoryDoesNotExist \\
\Xi FileSystem \\
n? : Name \\
report! : Message \\
\hline
cwd \frown \langle n? \rangle \notin dirs \\
report! = \text{“}Directory\ does\ not\ exist\text{”} \\
\hline
\end{array}
$$

4 Z MStix Specifications

We begin the specification with a description of the base types needed. There are two basic types of objects in the system: files and directories. The type *Name* corresponds to a simple file or directory name (for example, the MStix file "foo"):

$$[Name]$$

Sequences of *Name* are full file or directory names (for example, the MStix file "/fee/fie/foo"):

$$FullName ::= seq\, Name$$

The representation chosen here has the leaf elements at the tail of the sequence, and so, for example, the representation of "fee/fie/foo" is the sequence $\langle fee, fie, foo \rangle$. We use the Z sequence manipulation functions $front$, which yields a subsequence up to the last element and $last$, which yields the atom at the end of the sequence. All file names are unique in the system, as are full directory names. Further, we restrict the system so that a file and a directory may not share the same name.

4.1 State Description

The state of the file system is as follows:

FileSystem

$$files : \mathbb{P}\, FullName$$
$$dirs : \mathbb{P}\, FullName$$
$$cwd : FullName$$

$$\forall f : files \cup dirs \bullet f \neq \langle\rangle \Rightarrow front\, f \in dirs$$
$$dirs \cap files = \emptyset$$
$$cwd \in dirs$$

There are three components: $files$, $dirs$, and cwd. The first component, $files$, is the set of files that currently exist in the system. The second component, $dirs$, is the set of directories that currently exist in the system. The first invariant states that all intermediate directories must exist for a file or directory to exist. The second invariant states that file and directory names are distinct. The last component, cwd, does not record any permanent feature of the file system, but is instead used to mark a (user's) current directory in the system. The third invariant states that cwd must correspond to an existing directory.

We now proceed with operations that change or observe the state. We first specify the desired operation, and then proceed to make the operation total by defining behavior for cases in which the precondition of an operation is not satisfied.

4.2 The Initial State

A reasonable initial state for the file system is one in which there is only the root directory (i.e. the empty sequence):

InitFileSystem

$$FileSystem'$$

$$files' = \{\}$$
$$dirs' = \{\langle\rangle\}$$
$$cwd' = \langle\rangle$$

It is clear that the new state produced by $InitFileSystem$ ($FileSystem'$) satisfies the invariants of $FileSystem$.

- `DeleteDir` **DirName**
  If **DirName** is an empty subdirectory of the current directory, it is removed, otherwise an appropriate error message is printed.

- `Down` [**DirName**]
  If **DirName** exists as a subdirectory of the current, the current directory is set to **DirName**.

- `Up`
  The current directory is set to the parent of the current directory.

- `Root`
  The current directory is set to the root.

- `Where`
  Prints the "extended" name of the current directory. The extended name includes the name of every directory between the current directory and the root, inclusive.

- `CreateFile` **FileName**
  If **FileName** is not already a file or directory in the current directory, it is created in the current directory.

- `DeleteFile` **FileName**
  If **FileName** is a file in the current directory, it is removed, otherwise an appropriate error message is printed.

- `CopyFile` **OldFileName NewFileName**
  Creates a copy of the file **OldFileName** in **NewFileName** (since MSDOS ignores file contents, this is equivalent to creating a new file called **NewFileName**).

- `CopyFileDown` **OldFileName DirName**
  Copies **OldFileName** into the directory **DirName**.

- `CopyFileUp` **OldFileName**
  The file is copied to the parent of the current directory.

- `MoveFile` **OldFileName NewFileName**
  Changes the name of the file **OldFileName** to **NewFileName**.

- `MoveFileDown` **OldFileName DirName**
  Moves **OldFileName** into the directory **DirName**.

- `MoveFileUp` **OldFileName**
  The file is moved to the parent of the current directory.

- `ListDirs`
  Prints the names of all subdirectories in the current directory.

- `ListFiles`
  Prints the names of all files in the current directory.

- `Logoff`
  Stops execution.

MSDOS must handle the following exceptional conditions in a user friendly manner:

- Directory or file already exists on a create, move, or copy.
- Directory contains files on a remove.
- Directory does not exist on a change, or delete.
- File does not exist on a copy, move, or delete.

- `ChangeDir [NewDirName]`
  If *NewDirName* is given and exists as a subdirectory of the current, the current directory is set to *NewDirName*. If *NewDirName* is not given, the current directory is set to the root. If the *NewDirName* is "..", the current directory is set to the parent of the current directory.

- `Where`
  Prints the 'extended' name of the current directory. The extended name includes the name of every directory between the current directory and the root, inclusive.

- `CreateFile FileName`
  The file *FileName* is created in the current directory.

- `DeleteFile FileName`
  If *FileName* is in the current directory, it is removed, otherwise an appropriate error message is printed.

- `Move OldFileName NewFileName`
  `Move OldFileName NewDirName`
  Changes the name of the file *OldFileName* to *NewFileName* or moves *OldFileName* into the directory *NewDirName*. If the *NewDirName* is "..", the file is moved to the parent of the current directory.

- `Copy OldFileName NewFileName`
  `Copy OldFileName NewDirName`
  Creates a copy of the file *OldFileName* in *NewFileName* (since MStix ignores file contents, this is equivalent to creating a new file called *NewFileName*) or copies *OldFileName* into the directory *NewDirName*. If the *NewDirName* is "..", the file is copied to the parent of the current directory.

- `List`
  Prints the names of all files and subdirectories in the current directory. The subdirectories should be distinguished from the files by a trailing slash ("/").

- `Logoff`
  Stops execution.

MStix must handle the following exceptional conditions in a user friendly manner:

- Directory or file already exists on a create, move, or copy.
- Directory contains files on a remove.
- Directory does not exist on a change, or delete.
- File does not exist on a copy, move, or delete.

3 Revised Informal MStix Specifications

MStix is specified informally below with a description of the syntax and semantics of each command. Several of the commands take one or more arguments, which are directory or file names. Directory and file names are strings of characters, `'-'`, `'_'`, `'='`, and `'+'`.

- `InitFileSystem`
  Supply a valid initial (empty) state for the file system.

- `CreateDir DirName`
  If *DirName* is not already a file or directory in the current directory, creates a new directory called *DirName* as a subdirectory of the current directory, else an appropriate error message is printed.

1 Introduction

This paper presents several different specifications of a simple file system based on the Unix file system. This project was started from a class assignment initially used in undergraduate computer science courses at Clemson University, and more recently in graduate software engineering courses at George Mason University. In sections 2 and 3, we present two slightly different informal specifications for MStix. The initial specification was used in classes through Fall 1992. While deriving formal specifications and test specifications, we found several inconsistencies and inaccuracies with the specifications, as well as features that were difficult to express with formal approaches. The most important problems were that the initial specification does not completely describe file names, allows a file and a directory of the same name in the same subdirectory, and some of the original commands (`ChangeDir`, `CopyFile`, and `MoveFile`) actually combine several commands. We rectify these problems in the second informal specification and the formal specifications.

Section 4 gives a model-based specification for MStix in Z, and section 5 gives functional test specifications based on the Z specs and the Category-Partition method. Results from test cases derived from these specifications are in our companion paper [1]. Finally, section 6 gives an algebraic specification for MStix.

MStix is a hierarchical system consisting of *directories* and *files*. Each directory can contain an arbitrary number of files and *subdirectories*, where a subdirectory is also a directory. MStix ignores the contents of files; it is only concerned with the file's name. During operation, MStix keeps track of a "current directory". Initially, there is one empty directory called the *root*, which is the current directory.

There are a total of eighteen operations defined in (the revised specification for) MStix:

- One operation to create a valid (empty) initial state for the file system.
- Two operations to create and delete directories.
- Two operations to create and delete files.
- Three operations to copy files.
- Three operations to move files.
- Three operations to change the current directory.
- One operation to print the full pathname of the current directory.
- Two operations to list files and directories.
- One operation to log off.

2 Original Informal MStix Specifications

MStix is specified informally below with a description of the syntax and semantics of each command. Initially, there is one empty directory called the *root*, which is the current directory.

- `CreateDir` **DirName**
  If **DirName** is not already in the current directory, creates a new directory called **DirName** as a subdirectory of the current directory, else an appropriate error message is printed.
- `DeleteDir` **DirName**
  If **DirName** is an empty subdirectory of the current directory, it is removed, otherwise an appropriate error message is printed.

Functional and Test Specifications for the MiStix File System

Paul Amma nn
Jeff Offu tt

{pa mma n ,ofu t }@gmu .e du

De p a rt me n t o f In f o r ma t io n a n d So f t wa r e S yst e ms En
Ge o r g e Ma s o n Uni ve r s i t y, Fa i r f a x, VA 22 030
J a n u a r y 19, 1 9 9 3

Abstract

MiStix is a simple file system based on the Unix file system. MiStix is used in classroom exercises in graduate software engineering classes at George Mason University. In this document, we supply several different specifications for MiStix. First we give an informal specification. Next, since the informal specification turns out to be difficult to formalize directly, we supply a revised informal specification that matches subsequent formal specifications. We give a model-based specification for MiStix in Z, followed by functional test specifications based on the Category-Partition method. Finally, we give an algebraic specification for MiStix.