File System Example

This document contains a short plain text description (what to specify, important design decisions) and fragments of a possible solution (to illustrate some typical approaches).

Description

Individual files should have the following attributes: name, size, timestamp, and access privileges (for processes or users).

Files will be accessed either by processes or by users (or both?).

List of all operations:

- Create (new file)
- Destroy (remove file)
- Open file
- Close file
- Read (some data)
- Write (some data)
- Add (new data/chunk)
- Delete (some data)

Granularity of data access: chunks identified by some keys (or by their position and length), or individual characters at a given position (index).

Before reading or modifying a file, the user/process must open it. Design question: should we allow multiple processes to open the file for reading at the same time, or allow just exclusive access. Access for writing must be always exclusive.

Closing of a file will release it.

System state

First I show fragments of a specification for the system state.

Individual files may be specified in this way. Here we assume that they are made of chunks, where every chunk has a unique key.

Global basic data types:

$$[\, Key, Data \,]$$

Alternative: use *Index* or *Position* instead of *Key*.

We can model the contents of a file by a partial function.

File

$$contents : Key \pfun Data$$

The whole file system could be specified as follows. We need a global data type *Name*.

A key component of the system state is the set of currently open files.

FileSystem

$$files : Name \pfun File$$
$$openfiles : \mathbb{P}\, Name$$

$$openfiles \subseteq \mathrm{dom}\, files$$

We must also specify the initial content (empty) of a file and the initial state of the whole file system.

FileInit

$$File'$$

$$contents' = \varnothing$$

SystemInit

$$System'$$

$$files' = \varnothing$$

Additional basic global data types: *ProcessID*.

Operations

We have to define at least some operations.

The operation *OpenFile* has two parameters: a target file and process that accesses the file. Precondition (constraint): the given file is not yet open. Effect of the operation is change of the set of open files.

OpenFile

$$\Delta FileSystem$$
$$p? : ProcessID$$
$$f? : File$$

$$f? \notin \mathrm{dom}\, openfiles$$
$$openfiles' = openfiles \cup \{f?\}$$

Design choice: we could also track the process which opened the file. Then, *openfiles* would be a relation $\subseteq File \times ProcessID$.

A part of the specification of the operation *Create* will be the schema *FileInit*, because each new file must be initialized.

Operation local read of some data.

*Read*0

$$\Xi File$$
$$k? : Key$$
$$d! : Data$$

$$k? \in \mathrm{dom}\, contents$$
$$d! = contents(k?)$$

The operation does not modify the file. Input parameter is the key (and it must be valid). Output is the read data value.

Similarly, we can define the operation *Write*0 that rewrites an existing data chunk. It will change the function (mapping) *contents* in this way: $contents' = contents \oplus \{k? \mapsto d?\}$.

Operation *Add*0. Constraint: $k? \notin \mathrm{dom}\, contents$. Effect: $contents' = contents \cup \{k? \mapsto d?\}$.

Operation *Delete*0. Remove the given (key,data) pair from *contents*. Approach: $contents' = \{k?\} \lhd contents$ (drop $k?$ from the domain of the function *contents*).

Operations such as *Create* and *Destroy* may extend the schema *FileManage* that contains $\Delta FileSystem$, $n? : Name$, and the constraint $openfiles' = openfiles$.

Error handling

To properly model the action of opening a file, we must also cover the error situations. One possible approach is to define a schema for each error situation that may occur. The schema describes conditions under which the error occurs and also the proper response.

We will define schemas *AlreadyOpen* and *Unauthorized* (both just querying the current state). Constraints may look like this: $f? \in \mathrm{dom}\, openfiles$, $(f?, p?) \notin permissions$. Finally, we define a total operation $T\_OpenFile \mathrel{\widehat=} OpenFile \lor AlreadyOpen \lor Unauthorized$. This illustrates usage of the schema calculus.

Errors may occur also in the case of operations for reading and writing data — for example, "key not in use" or "index out of bounds".

We can introduce a global type *Report*.

$$Report ::= keyNotInUse \mid okay$$

And then we may define the schemas *KeyError*, *KeyNotInUse* and *Success*. This also illustrates how to include schemas (*KeyError* in *KeyNotInUse*).

KeyError

$\Xi File$
$k? : Key$
$r! : Report$

KeyNotInUse

$KeyError$

$k? \notin \mathrm{dom}\, contents$
$r! = keyNotInUse$
$contents' = contents$

Success

$r! : Report$

$r! = okay$

Full robust versions of operations such as read and write of/to some data chunk, including reports (success versus errors) are then defined as follows (using schema calculus).

$$Read \mathrel{\widehat=} (Read0 \land Success) \lor KeyNotInUse$$

Implementation normally cannot (and will not) separate these aspects (correct behavior, error handling). This is an example of abstraction possible when a high-level specification language is used.

Note that one could write specification of the robust version of the read operation directly, but that would be much less clear and readable. It is much better to create a modular specification.

For some of the operations we must also capture some additional error states: file exists, file does not exist, file is not open, etc.
A possible approach is to extend the data type *Report* (add some values) and introduce the schema *FileError* (similar to *KeyError*). Then we can introduce schemas like *FileExists* that will include *FileError*, assign particular values to the variable $r! : Report$, and define constraints that represent the corresponding situations. The really full operations *Read*, *Write*, *Add*, and *Delete* will be defined in this way:

$$FileRead \mathrel{\widehat=} Read \lor FileIsNotOpen \lor FileDoesNotExist$$

Also the file management operations, such as *Open*, *Close*, *Create*, and *Destroy*, will be specified like this:

$$Open \mathrel{\widehat=} (Open0 \land Success) \lor FileIsOpen \lor FileDoesNotExist$$

The general advantage of this approach, based on schema calculus, is that we can separately describe (1) the behavior for valid input and (2) error handling, and then combine all schemas into a complete specification.

This separation of normal behavior from error handling is just the basic (and most common) kind of modularization possible with schema calculus and Z.

An example of more complex modularization is framing. Operations defined for one entity, such as unnamed single file, are transformed into operations on a named entity in the whole file system. For example, one schema may specify reading of data from a given file, and another schema may describe access to a named file in some directory. Both schemas will be merged to get a specification for the operation of reading data from a named file. Specification of access restrictions (who is allowed to perform the given operation) may also be separated from the actual operation (what it does).