SPECIFYING SYSTEM IMPLEMENTATIONS IN Z

by

Jonathan Bowen
Roger Gimson
Stig Topp-Jørgensen

Technical Monograph PRG-63

February 1988

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Specifying System Implementations in Z

Jonathan Bowen
Roger Gimson
Stig Topp-Jørgensen

Abstract

In an introductory chapter, an outline is presented of some techniques for specifying the building of systems from subsystems using the formal notation Z. These techniques have been applied to the specification of implementations for services in a distributed system.

The major part of the monograph consists of an extended example showing how the implementation of a simple file server can be specified using some of the outlined techniques. The example file service is implemented in terms of a lower-level storage service. The specification includes the handling of errors that may arise because of this dependency.

Contents

Introduction

One of the most important steps in the implementation of any system of significant size is the clear identification of, and separation of the code into, a number of well-specified subsystems. Together, the subsystems implement the desired behaviour of the complete system, but each subsystem can be implemented separately as a system of its own.

This monograph is concerned with the specification of system implementations constructed from a number of separate subsystems. The specifications are expressed in the formal notation Z [1-4]. The first chapter discusses the use of Z to build systems from subsystems. The second chapter is an extended example showing how the implementation of a simple file service can be expressed using these techniques.

The Distributed Computing Software Project, from which this work arises, has been investigating the design, implementation and documentation of distributed system services using formal methods. Two earlier monographs present and discuss the general approach to service specification [5], and provide a larger example of the use of formal specification as a basis for the documentation of a service [6].

Some of the techniques presented in the first chapter have already been used in the earlier monographs. However, the file service implementation discussed in the second chapter is more complex and requires a different approach. In particular, it relies on a clear specification of the sequencing of suboperations within the implementation. Some of the extra complexity arises from the use of a separate lower-level storage service as part of the implementation. This then leads on to consideration of the behaviour of the implementation in the case of errors – which were ignored in the earlier examples.

Chapter 1

Building Systems from Subsystems in Z

1 Introduction

An important part of producing an implementation from a system design is the decomposition of the design into a number of separable subsystems, corresponding to separately coded parts of the implementation. It is from these subsystems that the complete implementation design is built.

In Z, the building of the state of a system from the states of a number of separate subsystems is relatively straightforward. Building the operations of the complete system from operations of the subsystems involves more detailed choices, particularly concerning the handling of parameters.

Few of the techniques described here are especially new (see [7] for some related examples). They have been used by Z practitioners in particular specifications for some time. However, no single document provides an overview of these techniques. The purpose of this chapter is to describe and contrast the techniques themselves, particularly when operation parameters are involved.

2 Conjoining states

Assume we wish to build a system from two existing subsystems. Let us call the subsystems A and B, and the combined system C.

Assume also that we already have models of the two subsystems in Z. In particular, the subsystems have state components S_A and S_B.

The state of the combined system S_C is simply the conjunction of the states of the subsystems.

$$S_C \mathrel{\widehat{=}} S_A \wedge S_B$$

Note that by use of schema conjunction we also cover the use of schema inclusion, so that the above definition could equally well have been written as follows.

```
S_C ───────┐
│   S_A
│   S_B
└──────────┘
```

If the subsystems have been separately specified, we will normally wish the components of their states to have disjoint names to eliminate unwanted superposition.

$$\mathrm{names}(S_A) \cap \mathrm{names}(S_B) = \emptyset$$

Though superposition of names in conjoined schemas has been found useful in some specifications, it implies 'sharing' of state components between the subsystems, which is contrary to good information hiding principles in conventional software design. We shall assume in the following that the names are disjoint. If necessary, this can be achieved by appropriate decoration or renaming of the original subsystems.

3 Operations

The operations which can be performed on the combined system will be built from the suboperations which are assumed to have already been defined on the subsystems.

Assume that we have operations P_A and Q_B defined on subsystems A and B respectively.

$$\begin{aligned} P_A &\mathrel{\widehat{=}} \Delta S_A \mid \ldots \\ Q_B &\mathrel{\widehat{=}} \Delta S_B \mid \ldots \end{aligned}$$

We assume ΔS and ΞS are defined in the conventional way unless otherwise specified. In particular, ΔS denotes an arbitrary change of state and ΞS denotes a change of state where the 'before' and 'after' states are the same:

$$\begin{aligned} \Delta S &\mathrel{\widehat{=}} S \wedge S' \\ \Xi S &\mathrel{\widehat{=}} \Delta S \mid \theta S' = \theta S \end{aligned}$$

3.1 Conjoining operations

The simplest way in which to specify an operation that can be applied to the combined system is to conjoin subsystem operations.

$$R_C \mathrel{\widehat{=}} P_A \wedge Q_B$$

This simply says that the effect of the combined operation R_C on S_C is the same as performing P_A on S_A and Q_B on S_B. Since the subsystem state components are disjoint, the operations on the subsystems may be thought of as being performed 'in parallel' (or, equally well, as being performed one after the other, in either order).

3.2 Operation parameters

In Z, parameters to operations are simply expressed as extra components (additional to those of the 'before' and 'after' state) in the signature of the operation schema. By convention, their names are written with a suffix of '?' or '!' indicating input and output parameters respectively. However, this does not provide a semantic difference from other schema components.

There is nothing to prevent the predicate in an operation schema from imposing a constraint on the value of an input parameter which may be incompatible with some values that might be supplied by the 'calling' environment. Similarly, there is nothing which forces the predicate to constrain an output parameter to have a particular value. (These freedoms are essential for allowing partial or non-deterministic specifications to be written.)

3.3 Disjoint parameters

It is simple to pass parameters to subsystem operations when the parameters are disjoint. Take the following suboperation definitions.

$$\begin{aligned} P_A &\mathrel{\widehat{=}} \Delta S_A;\ t?{:}T;\ u!{:}U \mid \dots \\ Q_B &\mathrel{\widehat{=}} \Delta S_B;\ v?{:}V;\ w!{:}W \mid \dots \end{aligned}$$

Since the parameters are non-interfering, the suboperations may be conjoined as before, giving a combined system operation with four parameters.

$$\begin{aligned} R_C &\mathrel{\widehat{=}} P_A \wedge Q_B \\ &= \Delta S_C;\ t?{:}T;\ v?{:}V;\ u!{:}U;\ w!{:}W \mid \dots \end{aligned}$$

The more common situation, however, is for there to be some sharing of parameters.

3.4 Shared parameters

The sharing may simply allow some parameters to be common inputs or common outputs to both suboperations. Take this example.

$$\begin{aligned} P_A &\mathrel{\widehat{=}} \Delta S_A{:}\ t?{:}T;\ u!{:}U \mid \dots \\ Q_B &\mathrel{\widehat{=}} \Delta S_B{:}\ t?{:}T;\ u!{:}U \mid \dots \end{aligned}$$

In other words, `t?` is input to both suboperations, and `u!` is output from both suboperations. Again the suboperations may be conjoined, with the parameter names becoming superposed.

$$\begin{aligned} R_C &\mathrel{\widehat{=}} P_A \wedge Q_B \\ &= \Delta S_C{:}\ t?{:}T;\ u!{:}U \mid \dots \end{aligned}$$

Some care is required to ensure that this combination remains meaningful. The precondition of R_C will be the conjunction of the preconditions of P_A and Q_B, which may mean that its domain is restricted. In particular, the combined operation will only be applicable if any constraints imposed on the value of t? by each suboperation are both satisfied.

With a shared output parameter, such as u! above, care must also be taken that the two operations do not simultaneously define incompatible output values. The most frequent use of such shared outputs is for something like a report value indicating the outcome of the operation. In this case, a common situation would be for each suboperation to define the output value for disjoint parts of the input domain.

3.5 Parameters between suboperations

Another form of sharing parameters is when some output produced by one suboperation is to be used as input to another suboperation. This normally implies that, in the implementation, the execution of the first suboperation must be completed before the second is started. Sequential composition of operation schemas, which we consider later, is the obvious way of combining the operations to reflect this ordering. However, a more abstract specification, which avoids overspecification of execution order, can be achieved with operation conjunction, if used with care.

Let us consider an example. Take the following suboperations.

$$P_A \;\widehat{=}\; \Delta S_A;\ u!:X \mid \ldots$$
$$Q_B \;\widehat{=}\; \Delta S_B;\ v?:X \mid \ldots$$

We wish the output of P_A to be the input to Q_B. This can be expressed using schema conjunction with renaming as follows.

$$R_C \;\widehat{=}\; P_A[x/u!] \wedge Q_B[x/v?]$$

The correspondence between parameters is achieved by renaming to a common intermediate name x (in preference to the asymmetric, and possibly confusing, alternative: $P_A \wedge Q_B[u!/v?]$).

The intermediate x may be considered as simply a device for describing the parameter correspondence, in which case it should probably be hidden to avoid clashes in subsequent operation combinations.

$$R_C \;\widehat{=}\; (P_A[x/u!] \wedge Q_B[x/v?]) \setminus (x)$$

3.6 Parameter passing cycles

Care must be taken not to introduce unimplementable parameter passing structures containing cycles, the most trivial example being where each suboperation of a pair provides an output which is input to the other.

$$\begin{aligned} P_A &\mathrel{\widehat{=}} \Delta S_A;\ t?:Y;\ u!:X \mid \ldots \\ Q_B &\mathrel{\widehat{=}} \Delta S_B;\ v?:X;\ w!:Y \mid \ldots \\ R_C &\mathrel{\widehat{=}} P_A[y/t?, x/u!] \land Q_B[x/v?, y/w!] \qquad ??? \end{aligned}$$

Though there may be nothing wrong with such a non-constructive specification, it means that an implementation cannot be constructed by any meaningful combination of the two subsystem operations to satisfy this specification.

3.7 Piped parameters

A special schema notation has been suggested for use in situations where parameters are passed from one suboperation to the next in the style of a pipeline.

$$R_C \mathrel{\widehat{=}} P_A[x!/u!] \gg Q_B[x?/v?]$$

The renaming is slightly different from the conjoined case, since piping superposes only matching '!' and '?' parameters, and it includes the hiding of such matched parameters. With suitable initial choice of parameter names in the subsystems the renaming would be unnecessary.

4 Homogeneous operations

A wider range of operation constructors becomes applicable if the operations to be composed are homogeneous; in other words, if they are defined over the same state rather than each being defined on a different subsystem state.

We can redefine the subsystem operations to apply to the combined system state, ensuring in each case that the other subsystem does not change.

$$
\begin{aligned}
P_C &\mathrel{\widehat{=}} P_A \land \Xi S_B \\
Q_C &\mathrel{\widehat{=}} Q_B \land \Xi S_A
\end{aligned}
$$

This could also be written in a more general form, avoiding explicit mention of the unchanging subsystem (useful when there are several subsystems conjoined in the combined state), through the use of schema hiding.

$$
\begin{aligned}
P_C &\mathrel{\widehat{=}} P_A \land \Xi S_C \backslash \Delta S_A \\
Q_C &\mathrel{\widehat{=}} Q_B \land \Xi S_C \backslash \Delta S_B
\end{aligned}
$$

4.1 Disjoining operations

Once homogeneous forms of the subsystem operations have been defined, schema disjunction becomes available as an operation combinator.

$$
R_C \mathrel{\widehat{=}} P_C \lor Q_C
$$

In general, this specifies a non-deterministic choice between performing one suboperation or the other, leaving the remaining subsystem unchanged.

However, in the situation that the preconditions of the two suboperations are mutually exclusive, only one of the suboperations is applicable for a given set of input values and the choice reduces to a deterministic one.

4.2 Overriding operations

A special case of disjoining operations is when the precondition of Q_C is used to determine the choice between P_C and Q_C. In this case the special notation of schema overriding becomes applicable.

$$\begin{aligned} R_C &\mathrel{\widehat{=}} P_C \oplus Q_C \\ &= (P_C \land \neg \mathrm{pre}\ Q_C) \lor Q_C \end{aligned}$$

This is most frequently encountered in the specification of exceptional conditions, or error reports, where the precondition of Q_C is an error condition that must be false for the 'normal' operation P_C to succeed, and where otherwise Q_C is used to define the outcome in the error case.

5 Sequencing operations

The next form of operation combinator can be seen as introducing a more concrete view of the construction of systems. In particular, it introduces the idea of operation sequencing, so that one suboperation is explicitly specified as following another. In this sense, it forms the first step towards more implementation-oriented specifications.

5.1 Composing operations

Schema composition can be used to combine homogeneous subsystem operations. The combined operation is then written in an explicitly 'sequential' form.

$$R_C \mathrel{\widehat{=}} P_C \mathbin{;} Q_C \qquad \text{or} \qquad R_C \mathrel{\widehat{=}} Q_C \mathbin{;} P_C$$

If there is no 'communication' between the two subsystems (because they affect different parts of the state and they have no parameters), the ordering is unimportant and both alternatives would reduce to the same net effect as the conjunction $P_A \wedge Q_B$. Things get more complicated, however, when parameters are introduced.

The same technique of using a hidden variable (as in section 4.5) can be used, with the advantage that composition makes it easier to see that the suboperation which sets the value of the variable must precede the suboperation which makes use of that value.

$$R_C \mathrel{\widehat{=}} (P_C[x/u!] \mathbin{;} Q_C[x/v?]) \setminus (x)$$

5.2 Parameter buffers

An alternative to having a hidden variable to show parameter correspondence is to explicitly include a 'parameter buffer' in the combined system state.

$$XBuf \mathrel{\widehat{=}} [x : X]$$
$$S_C \mathrel{\widehat{=}} S_A \wedge S_B \wedge XBuf$$

Each suboperation is extended to explicitly set, or leave unchanged, the value of this buffer, as well as to leave other subsystems unchanged. Any operation may make use of the current value of the buffer.

$$\begin{aligned}
P_C &\mathrel{\widehat{=}} P_A[x'/u!] \wedge \Xi S_C \setminus \Delta S_A \setminus \Delta XBuf \\
Q_C &\mathrel{\widehat{=}} Q_B[x\ /v?] \wedge \Xi S_C \setminus \Delta S_B \\
R_C &\mathrel{\widehat{=}} P_C \mathbin{;} Q_C
\end{aligned}$$

Here, P_C sets the value of the buffer, while Q_C leaves it unchanged but makes use of its current value. (Both definitions include ΞS_C, which includes $\Xi XBuf$, but in P_C change is allowed because $\Delta XBuf$ is hidden from ΞS_C).

This use of buffers is clearly closer to an implementation-oriented description, in which the buffer may be seen as a programming language variable that will retain its value unless explicitly changed by an assignment (i.e. value-changing operation). Note that in Z, it is necessary to explicitly state that the value will be left unchanged by some operations (or, as in the form above, to specify that an operation may change its value by hiding $\Delta XBuf$ from the ΞS_C schema).

6 Programming in Z

In order to construct more complex operations from suboperations, particularly when specifying an implementation-oriented view of a system, it is often useful to use the kind of constructors found in conventional programming languages.

Sequential composition of operations has already been considered. Here we introduce definitions for conditional, iterative and interleaving constructors.

Further discussion of the transformation of Z specifications into programs can be found in [8-10].

In the following, we will assume that P and Q are homogeneous operations on a state S (having undashed and dashed components representing the state before and after the operation), and B is a schema representing a predicate defined only on the current state (involving no change of state).

6.1 Conditional

$$P \ \underline{\mathrm{if}} \ B \ \underline{\mathrm{else}} \ Q \quad \widehat{=} \quad (B \wedge P) \vee (\neg B \wedge Q)$$

or, if there is no 'else' part

$$P \ \underline{\mathrm{if}} \ B \quad \widehat{=} \quad (B \wedge P) \vee (\neg B \wedge \Xi S)$$

6.2 Iteration

Let

$$I_0 \quad \widehat{=} \quad \neg B \wedge \Xi S$$
$$I_{i+1} \quad \widehat{=} \quad (B \wedge P) \mathbin{⨾} I_i \qquad \forall\, i : \mathbb{N}$$

then

$$P \ \underline{\mathrm{while}} \ B \quad \widehat{=} \quad I_0 \vee I_1 \vee I_2 \vee \ldots$$

6.3 Interleaving

$$P \parallel Q \;\widehat{=}\; (P \fatsemi Q) \vee (Q \fatsemi P)$$

This may be generalised to $\parallel_{n:N} P$, where n is a component in the schema P, and N is a set compatible with n's type.

If N is the empty set,

$$\parallel_{n:N} P \;\widehat{=}\; \equiv S$$

otherwise, if N is not empty, $\parallel_{n:N} P$ represents the logical disjunction of all possible sequences of P, each with different values of n chosen from N and with n hidden. For example, if $N \widehat{=} \{1, 2, 3\}$, then:

$$\begin{aligned}\parallel_{n:N} P \;\widehat{=}\; & (P_1 \fatsemi P_2 \fatsemi P_3) \vee (P_2 \fatsemi P_1 \fatsemi P_3) \vee (P_2 \fatsemi P_3 \fatsemi P_1) \vee \\ & (P_1 \fatsemi P_3 \fatsemi P_2) \vee (P_3 \fatsemi P_1 \fatsemi P_2) \vee (P_3 \fatsemi P_2 \fatsemi P_1)\end{aligned}$$

where $P_1 \widehat{=} (P \mid n=1) \setminus (n)$; $P_2 \widehat{=} (P \mid n=2) \setminus (n)$; $P_3 \widehat{=} (P \mid n=3) \setminus (n)$

7 Conclusion

The techniques presented in this chapter have heen used in various specifications produced as part of the Distributed Computing Software Project.

The use of schema conjunction, disjunction and overriding to define an operation from constituent parts is commonplace in most Z specifications of system components. Examples of their use, and particularly of overriding to define error behaviour, can be found in [5].

The Block Storage Service Implementor Manual (contained in [6]) illustrates the building of systems from subsystems in which the implementations of the operations on the service make use of conjoined suboperations (denoted by schema inclusion). The parameter passing techniques described in section 3.4 are used to pass parameters between the suboperations within an operation implementation.

The following chapter in this monograph illustrates the techniques described in section 5, making use of suboperation sequencing, and explicitly including parameter buffers as part of the state of the implemented system.

Chapter 2

Implementing a simple File Service

1 Introduction

To illustrate some of the methods described in the previous chapter, we shall now consider the implementation of a simple file service, the PageFile Service, so called because each file consists of a (possibly sparse) array of fixed-size pages of data.

First the user's view of the service is presented with just sufficient detail to give a precise definition of what is to be implemented. Then two simple subsystems, a Page Store and a Header Store, are identified and the abstract states of these subsystems are formally defined. The concrete state of the PageFile Service is defined in terms of the abstract states of the subsystems and some constraints on the concrete service state are proposed to ensure the efficiency of the implementation. Also, the representation relation between the abstract and concrete state of the PageFile Service is formally defined.

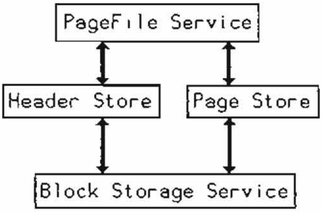
The next step is to consider the implementation of the concrete service operations. For the sake of simplicity only the successful outcome of operations are considered initially. The successful operations on the two abstract subsystems are specified and some further auxiliary operations (not affecting the state of the subsystems) are defined. The successful cases of the concrete operations on the file service are then specified in terms of the operations on the subsystems and the auxiliary operations.

Some of the abstract service operations have errors associated with them. To allow the concrete service specification to mirror this, the abstract specifications of the subsystems and the auxiliary operations are modified to take such errors into account. The concrete operations on the file service are then redefined in terms of these operations.

By this stage, the subsystem operations have been specified as ideal in the sense that they always will return a predictable result. However, if these subsystems are to be implemented in terms of other services, possibly residing on other hosts in the distributed system, the implementation must allow for errors such as the crash of a subsystem and network failures. The specifications of the subsystems are modified to allow such errors to occur. The effect of these changes on the specifications of the concrete PageFile Service operations is studied. It is shown how consideration of such errors is to some extent incompatible with the efficiency constraints stated earlier.

Finally the implementations of the Page Store and the Header Store are both specified in terms of the Block Storage Service described in [6].

The structure of the implementation can be illustrated as:



The PageFile Service is implemented in terms of the Header Store and the Page Store each of which in turn is implemented in terms of the Block Storage Service.

The final design presented in this chapter is not directly implementable, but is detailed enough for a competent programmer to implement in a chosen imperative programming language with a minimum of effort.

2 User view of the service

In this section we shall present the user's view of the PageFile Service in abbreviated form. Only sufficient detail is included to give an unambiguous description of what is to be implemented. The full User Manual follows the style of the Block Storage Service (see [6]).

The PageFile Service provides data storage facilities for *pagefiles* consisting of a set of numbered fixed-size pages. It is intended as a simple intermediate service on top of which more elaborate files (such as files consisting of arbitrary-length sequences of bytes) could be implemented.

Pagefiles may be created, updated, accessed and destroyed by clients. An identifier, chosen by the service, is used to identify a particular pagefile. A unique identifier is given to each pagefile, a new identifier being issued each time a pagefile is changed. Pagefiles have a limited lifetime, with an expiry time chosen by the client, and will be destroyed without warning on reaching the given expiry time.

The service provides limited security of access to pagefiles. A client may not access a pagefile without knowing its identity, and pagefile identifiers are hard to guess (since their values are chosen from a very large set). The identity of any pagefile is initially known only to its creator; the service will never tell the identity of a pagefile to any other client. Pagefiles may be updated or destroyed only by their creators, and so security also depends on the proper authentication of clients.

Implementation-specific constants, which are also not defined further, are shown in italics (e.g. *PageSize*). The following basic sets are also used:

$$[\mathrm{UserNum}, \mathrm{Time}, \mathrm{Report}, \mathrm{Id}, \mathrm{Byte}]$$

2.1 PageFiles

The PageFile Service stores pagefiles on behalf of its clients. A pagefile is a file consisting of an indexed set of *pages*. Each page is a fixed size array of bytes.

$$\mathrm{Page} \mathrel{\widehat{=}} 0\,..\,(\mathit{PageSize}-1) \rightarrow \mathrm{Byte}$$

Pages in a pagefile are numbered in sequence.

$$\mathsf{PageNum} \mathrel{\widehat{=}} 0\,..\,(MaxPages-1)$$

The data in a pagefile consists of the numbered pages. Not every number need have an associated page of data at any particular moment.

$$\mathsf{PageFileData} \mathrel{\widehat{=}} \mathsf{PageNum} \nrightarrow \mathsf{Page}$$

As well as containing the client's data, the pagefile records some general information: the *owner* of the pagefile (the identity of the client who created it), the time of its creation, the time of its last update and the time of its expiry.

$$\begin{array}{|l}
\textsf{Info} \\
\hline
\quad \mathsf{owner} : \mathsf{UserNum} \\
\quad \mathsf{created} : \mathsf{Time} \\
\quad \mathsf{updated} : \mathsf{Time} \\
\quad \mathsf{expires} : \mathsf{Time} \\
\hline
\quad \mathsf{created} \leqslant \mathsf{updated} \\
\quad \mathsf{created} \leqslant \mathsf{expires} \\
\hline
\end{array}$$

Whenever a pagefile is created, an *expiry time* must be given by the client; it is the time until which the service is obliged to store the pagefile. On reaching its expiry time, a pagefile is said to have *expired*, and can be discarded by the service without notification of the client. A pagefile consists of the information above and its data.

$$\mathsf{PageFile} \mathrel{\widehat{=}} \mathsf{Info}\ ;\ \mathsf{data} : \mathsf{PageFileData}$$

An *id* (identifier) will be issued by the service when the pagefile is created, taken from the set Id of all identifiers. This becomes the client's reference to the pagefile and any subsequent operations on the pagefile will require this identity.

2.2 Service state

The service state records all currently stored pagefiles according to their identities. It also contains a finite set of new pagefile ids which have not yet been issued. When a new id is issued, it is taken from this set. The schema PFS denotes the state of the PageFile Service at any particular moment.

```
PFS ──────────────────────────────────
│  files   : Id ⇸ PageFile
│  newids  : 𝔽 Id
├──────────────
│  newids ∩ dom files = ∅
│  NullId ∉ newids ∪ dom files
└──────────────────────────────────────
```

The service guarantees never to issue the special identity *NullId*; this id can therefore be used by clients' applications to indicate "no file".

Initially, when the service is started for the first time, there are no stored pagefiles, and all ids except the *NullId* are available.

```
InitPFS ────────────────────────
│  PFS′
├──────────────
│  files′  = ∅
│  newids′ = Id \ {NullId}
└───────────────────────────────
```

2.3 Parameters

The general aspects of operations on the PageFile Service, including the client number, current time and an output report are combined in the following schema, relating the state of the service before an operation (PFS) to that after the operation (PFS′).

```
ΔPFS ──────────────────────────────────
│  PFS
│  PFS′
│  clientnum : UserNum
│  now       : Time
│  report!   : Report
├──────────────
│  newids′ = newids \ dom files′
└──────────────────────────────────────
```

It is a property of every operation that any id issued by it is removed from the set of new ids, and so can never be issued again. Sometimes the state of the PageFile Service is left unaffected by an operation, particularly if an error is detected.

$$\Xi PFS \mathrel{\widehat{=}} \Delta PFS \mid \theta PFS' = \theta PFS$$

2.4 PageFile-specific operations

Many operations on the service apply to an existing pagefile stored by the service, and require the id of this pagefile to be supplied as an input parameter by the client. A *framing* schema is used to include this information in a specific operation definition.

```
ΦPageFile ________________
|   ΔPFS
|   file : PageFile
|   id? : Id
|_______________
|
|   file = files(id?)
|________________________
```

The `PageFile` stored under the given id (if one exists) is made an implicit parameter of such operations. Similarly, some operations produce a new pagefile and store it in the service, returning its new id as an output parameter. Such a pagefile is always owned by the current client and its update time is the current time (its creation and expiry time, and data, will be given in the particular operation definition). Its id is taken from the set of new ids. This is denoted by another framing schema.

```
ΦNewPageFile ________________________
|   ΔPFS
|   newfile : PageFile
|   id! : Id
|_______________
|
|   newfile.owner   = clientnum
|   newfile.updated = now
|   id! ∈ newids
|   newfile = files'(id!)
|____________________________________
```

2.5 Error reports

The report! output parameter of each operation indicates either that the operation succeeded or suggests why it failed. In all cases, failure leaves the state of the service unchanged. Success indicates successful completion of the operation.

$$\mathrm{Success} \mathrel{\widehat{=}} [\ \mathrm{report!} : \mathrm{Report} \mid \mathrm{report!} = \mathit{SuccessReport}\ ]$$

The total effect of a service operation is in general defined by *overriding* the definition of the successful outcome of the operation by one or more error report schemas. If the precondition in the error schema is satisfied, the corresponding error report is returned. Only if the precondition is not satisfied (usually corresponding to the satisfaction of a precondition in the successful operation definition) may the operation succeed.

NoSuchFile is given if there is no pagefile stored with identity id?.

```
NoSuchFile ______________________________
│   ΞPFS
│   id? : Id
├────────────────
│   id? ∉ dom files
│   report! = NoSuchFileReport
└────────────────────────────────────────
```

NoSuchPage denotes that there is no page with number pn? in pagefile id?.

```
NoSuchPage ______________________________
│   ΞPFS
│   ΦPageFile
│   pn? : PageNum
├────────────────
│   pn? ∉ dom file.data
│   report! = NoSuchPageReport
└────────────────────────────────────────
```

NoSpace indicates that a new pagefile cannot be created when the storage capacity of the service is exhausted. The service capacity is not modelled explicitly here, and so this error may occur non-deterministically, but it is guaranteed that the state of the service will be unaffected in this case.

```
NoSpace
    ΞPFS
    nospace : Boolean
    ─────────────────
    nospace = True
    report! = NoSpaceReport
```

NotOwner indicates an attempt to perform an operation which can destroy a pagefile by someone other than the owner of the pagefile.

```
NotOwner
    ΞPFS
    ΦPageFile
    ─────────────────
    file.owner ≠ clientnum
    report! = NotOwnerReport
```

2.6 Service operations

On the following pages appear descriptions of the service operations. Additionally, the following operation may be invoked at any time to remove expired files.

```
Scevenge
    ΔPFS
    expired : 𝔽 Id
    ─────────────────
    expired = {f:dom files | (files f).expires ≤ now}
    files' = expired ⩤ files
```

CREATEFILE

Abstract

```
CreateFile ( expires? : Time;
             pn?      : PageNum;
             page?    : Page;
             id!      : Id;
             report!  : Report )
```

A new pagefile is created with a specified expiry time, and is stored by the service under the new pagefile id!. The pagefile contains one page having the given page number and data.

Definition

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CreateFile}_{success}} \\
\quad \Delta PFS \\
\quad expires? : Time \\
\quad pn? \quad\quad : PageNum \\
\quad page? \quad : Page \\
\quad \Phi NewPageFile \\
\hline
\quad newfile.created = now \\
\quad newfile.expires = \max \{expires?, now\} \\
\quad newfile.data \quad = \{pn? \mapsto page?\} \\
\quad files' = files \cup \{id! \mapsto newfile\} \\
\hline
\end{array}
$$

The owner of the pagefile is the client. If an expiry time in the past is given, then the expiry time of the pagefile is set to now.

A new identifier is chosen which has never before been issued, and the new pagefile is stored under that id.

Reports

$$
\begin{array}{lll}
CreateFile & \widehat{=} & (CreateFile_{success} \wedge Success) \\
 & & \qquad\qquad\qquad\quad \oplus\ NoSpace
\end{array}
$$

WRITEFILE

Abstract

```
WriteFile ( id?     : Id;
            pn?     : PageNum;
            page?   : Page;
            id!     : Id;
            report! : Report )
```

An existing pagefile with the given id? is replaced by a new pagefile with a new id! which has the new page? at the specified pn?. The old pagefile is *destroyed*.

Definition

```
┌ WriteFile_success ──────────────────────────────────
│   ΔPFS
│   ΦPageFile
│   pn?   : PageNum
│   page? : Page
│   ΦNewPageFile
├───────────────
│   newfile.created = file.created
│   newfile.expires = file.expires
│   newfile.data    = file.data ⊕ {pn? ↦ page?}
│   files' = (id? ⩤ files) ∪ {id! ↦ newfile}
└─────────────────────────────────────────────────────
```

The creation and expiry times of the new pagefile are the same as the original pagefile. Only the owner may write to a pagefile.

A new id is chosen which has never previously been issued, and the new pagefile is stored under that id. The old pagefile is removed from the service.

Reports

```
WriteFile ≙ (WriteFile_success ∧ Success)
                               ⊕ NoSpace
                               ⊕ NotOwner
                               ⊕ NoSuchFile
```

READFILE

Abstract

```
ReadFile ( id?     : Id;
           pn?     : PageNum;
           page!   : Page;
           report! : Report )
```

The page with the specified pn? in the pagefile called id? is returned.

Definition

```
ReadFile_success ──────────────┐
│   ΞPFS
│   ΦPageFile
│   pn?   : PageNum
│   page! : Page
├──────────────────
│   page! = file.data pn?
└──────────────────────────────┘
```

The service is unchanged by this operation.

Any client may read a pagefile if they know its pagefile id.

An error report is produced if the pagefile does not have a page of data with the given page number.

Reports

$$
\begin{aligned}
ReadFile \mathrel{\widehat{=}} (ReadFile_{success} & \land Success) \\
& \oplus NoSuchPage \\
& \oplus NoSuchFile
\end{aligned}
$$

DESTROYFILE

Abstract

```
DestroyFile ( id?     : Id;
              report! : Report )
```

The pagefile stored under `id?` is removed from the service.

Definition

```
DestroyFile_success ─────────────────┐
│   ΔPFS
│   ΦPageFile
├──────────────────────
│   files′ = {id?} ⩤ files
└─────────────────────────────────────┘
```

A pagefile may be destroyed only by its owner.

Reports

```
DestroyFile  ≙  (DestroyFile_success ∧ Success)
                                     ⊕ NotOwner
                                     ⊕ NoSuchFile
```

SETFILEEXPIRY

Abstract

```
SetFileExpiry ( id?      : Id;
                expires? : Time;
                id!      : Id;
                report!  : Report )
```

An existing pagefile stored under `id?` is replaced by a new pagefile with a new `id!` and a new expiry time, but having the same data. The old pagefile is *destroyed*.

Definition

```
SetFileExpiry_success ─────────────────────────────
│  ΔPFS
│  ΦPageFile
│  expires? : Time
│  ΦNewPageFile
├──────────────────
│  newfile.created = file.created
│  newfile.expires = max {expires?, now}
│  newfile.data    = file.data
│  files' = (id? ⩤ files) ∪ {id! ↦ newfile}
└──────────────────────────────────────────────────
```

The new pagefile has the same data and creation time as the old pagefile. The client must be the owner of the file.

If an expiry time in the past is given, then the expiry time of the pagefile is set to `now`.

Reports

```
SetFileExpiry ≙ (SetFileExpiry_success ∧ Success)
                                       ⊕ NotOwner
                                       ⊕ NoSuchFile
```

3 Implementation subsystems

In order to determine the concrete state and the corresponding operations on it, we need to determine the subcomponents of that state.

An obvious choice of subsystems is a page store to hold the data contents of the files and a header store to hold the remaining information, including an index to the data pages.

3.1 Page Store

The Page Store allows a user to create, retrieve and destroy pages. When a page is created the Page Store assigns a unique PageId to it. This id is then used in all future references to that page. A special identifier, the *NullPageId*, is reserved for special purposes and will never be issued.

Together with the actual contents of a page, the Page Store will record its expiry time.

```
PageInfo ____________________
|   expires  : Time
|   contents : Page
|____________________________
```

The state of the Page Store can be defined as:

```
PS ______________________________________________
|   pages : PageId ⇸ PageInfo
|   newpageids : 𝔽 PageId
|_______________________
|   newpageids ∩ dom pages = ∅
|   NullPageId ∉ newpageids ∪ dom pages
|________________________________________________
```

The state records all currently stored pages according to their identities. It also maintains a set of page ids which have not yet been issued.

Initially, when the service is started, there are no stored pages and all page ids except the *NullPageId* are potentially available for issue.

```
InitPS ______________________________
|   PS'
|_______________
|
|   pages' = ∅
|   newpageids' = PageId \ {NullPageId}
|____________________________________
```

The Page Store as described here is very similar to the Block Storage Service [6]. Indeed we shall later see that it is a quite trivial matter to implement the Page Store in terms of the Block Storage Service.

3.2 Header Store

The contents of a pagefile can be described in terms of a contiguously numbered array of PageIds (corresponding to pages stored by the Page Store).

$$\mathsf{PageSeq} \;\widehat{=}\; \mathsf{PageNum} \rightarrow \mathsf{PageId}$$

A special case is the representation of the empty file:

$$\mathsf{EmptySeq} \;\widehat{=}\; \{\, \mathsf{s{:}PageSeq} \mid \mathsf{ran\ s} = \{NullPageId\} \,\}$$

Assuming that the actual pages will be held in the Page Store, a pagefile can be adequately represented by its "header":

```
Header ______________________
|   Info
|   filecontents : PageSeq
|____________________________
```

Using the new file representation the state of the Header Store can be defined as:

```
HS ________________________________________
|  headers       : Id ⇸ Header
|  newheaderids  : 𝔽 Id
|__________________________
|  newheaderids ∩ dom headers = ∅
|  NullId ∉ newheaderids ∪ dom headers
|__________________________________________
```

The state records all currently stored headers according to their identities and maintains a set of ids which have not yet been issued.

Initially, when the service is started, there are no stored headers and all file ids except the *NullId* are potentially available for issue.

```
InitHS ____________________________
|  HS'
|_____________
|  headers' = ∅
|  newheaderids' = Id \ {NullId}
|__________________________________
```

3.3 Combined state

The concrete state of the entire PageFile Service can be expressed by combining the two subsystems:

```
cPFS ________________________________________
|  PS
|  HS
|  now : Time
|_____________
|  ∀ pf:ran headers •
|       pf.expires > now ⇒
|         ∀ p:ran pf.filecontents •
|              p = NullPageId ∨
|              (pages p).expires ≥ pf.expires
|____________________________________________
```

The page ids contained in a non-expired header are those of the *NullPageId* and of pages stored in the Page Store. A page must not expire before the header from which it is referenced.

For the sake of the efficiency of the implementations we should ideally like to impose some further constraints.

A page expires at the same time as the header from which it is referenced.

```
ExpiryConstraint
    cPFS
    ──────────
    ∀ pf:ran headers •
       pf.expires > now ⇒
          ∀ p:ran pf.filecontents \ {NullPageId} •
             (pages p).expires = pf.expires
```

At any given time, the Page Store will only hold pages which are referenced from headers stored in the Header Store.

```
CompactnessConstraint
    cPFS
    ──────────
    ∀ pid:dom pages •
       ∃ pf:ran headers •
           pid ∈ ran pf.filecontents
```

We shall later see that these constraints are incompatible with other requirements of the service, and they are therefore not a mandatory part of the final specification.

3.4 Representation relation

The relation between the abstract and concrete representation of the PageFile Service can be defined as:

```
RelPFS
    PFS
    cPFS

    dom files = dom headers
    newids = newheaderids
    ∀ pf:dom files •
      f.expires > now ⇒
        f.owner   = h.owner   ∧
        f.created = h.created ∧
        f.updated = h.updated ∧
        f.expires = h.expires ∧
        f.filecontents =
            h.filecontents ⩥ {NullPageId} ⨾ pages
      where
        f ≙ (files pf)
        h ≙ (headers pf)
```

For each file in the abstract representation there is a header in the concrete representation. The file information is stored directly in the header. The contents of the file may be found by retrieving the pages whose non-null ids are stored in the header.

4 Successful operations

This section concentrates on describing the successful behaviour of the concrete operations whose abstract equivalents were described in section 2.

First a number of suboperations will be defined. These consist of operations on the two subsystems and a few auxiliary operations. Then it is shown how the service operations can be described in terms of these suboperations. For the moment, it is assumed that the suboperations are always successful.

4.1 Abstract operations on the Page Store

Create a new page and return the id of that page.

```
CreatePage_success ─────────────────────────────────┐
│   ΔPS
│   page?     : Page
│   expires?  : Time
│   pageid!   : PageId
│   pageinfo  : PageInfo
├──────────────────────────
│   pageinfo.expires = expires?
│   pageinfo.contents = page?
│   pageid! ∈ newpageids
│   pages' = pages ∪ {pageid! ↦ pageinfo}
│   newpageids' = newpageids \ {pageid!}
└───────────────────────────────────────────────────┘
```

Read an existing page.

```
ReadPage_success ───────────────────────────────┐
│   ΞPS
│   pageid? : PageId
│   page!   : Page
├──────────────────────────
│   pageid? ∈ dom pages
│   page! = (pages pageid?).contents
└───────────────────────────────────────────────┘
```

Destroy an existing page.

```
DestroyPage_success
    ΔPS
    pageid? : PageId
────────────────────
    pageid? ∈ dom pages
    pages′ = {pageid?} ⩤ pages
    newpageids′ = newpageids
```

Change the expiry date of a page, leaving the other page information (including page id) unchanged.

```
SetPageExpiry_success
    ΔPS
    now : Time
    pageid?  : PageId
    expires? : Time
────────────────────
    pageid? ∈ dom pages
    {pageid?}⩤pages′ = {pageid?}⩤pages
    (pages′ pageid?).contents =
                        (pages pageid?).contents
    (pages′ pageid?).expires = max {now, expires?}
    newpageids′ = newpageids
```

The Page Store is automatically scavenged periodically and expired pages removed.

```
ScavengePages
    ΔPS
    now : Time
────────────────────
    pages′ = {p:dom pages |
                 (pages p).expires > now} ⩤ pages
    newpageids′ = newpageids
```

4.2 Abstract operations on Header Store

The operations which may be performed on the Header Store are very similar to those for the Page Store.

Create a new header and return its id.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{CreateHeader}_{success}} \\
\quad \Delta HS \\
\quad header? : Header \\
\quad id! \quad\;\; : Id \\
\hline
\quad id! \in newheaderids \\
\quad headers' = headers \cup \{id! \mapsto header?\} \\
\quad newheaderids' = newheaderids \setminus \{id!\} \\
\hline
\end{array}
$$

Read an existing header.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{ReadHeader}_{success}} \\
\quad \Xi HS \\
\quad id? \quad\;\; : Id \\
\quad header! : Header \\
\hline
\quad id? \in \mathrm{dom}\ headers \\
\quad header! = headers\ id? \\
\hline
\end{array}
$$

Replace an existing header. The old header is deleted and a new one created (with a new id).

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\mathrm{ReplaceHeader}_{success}} \\
\hline
\Delta HS & \\
id? & : Id \\
header? & : Header \\
id! & : Id \\
\hline
\multicolumn{2}{|l}{id? \in \mathrm{dom}\ headers} \\
\multicolumn{2}{|l}{id! \in newheaderids} \\
\multicolumn{2}{|l}{headers' = (\{id?\} ⩤ headers) \cup \{id! \mapsto header?\}} \\
\multicolumn{2}{|l}{newheaderids' = newheaderids \setminus \{id!\}} \\
\hline
\end{array}
$$

Destroy an existing header.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{DestroyHeader}_{success}} \\
\hline
\Delta HS \\
id? : Id \\
\hline
id? \in \mathrm{dom}\ headers \\
headers' = \{id?\} ⩤ headers \\
newheaderids' = newheaderids \\
\hline
\end{array}
$$

The Header Store is scavenged periodically and expired headers removed.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{ScavengeHeaders}} \\
\hline
\Delta HS \\
now : Time \\
\hline
headers' = \{\ hd : \mathrm{dom}\ headers \mid \\
\qquad\qquad (headers\ hd).expires > now\ \} \lhd headers \\
newheaderids' = newheaderids \\
\hline
\end{array}
$$

4.3 Auxiliary operations

Apart from the operations on the subsystems, a number of other suboperations are needed.

Create an empty fileheader (i.e. for a file without any pages)

```
MakeHeader ___________________________
|   expires? : Time
|   header!  : Header
|   clientnum : UserNum
|   now : Time
|______________________
|   header!.owner   = clientnum
|   header!.created = now
|   header!.updated = now
|   header!.expires = expires?
|   header!.filecontents = EmptySeq
|_____________________________________
```

Extract from a header the page id corresponding to a given page number.

```
GetPageId _____________________________
|   header? : Header
|   pn?     : PageNum
|   pageid! : PageId
|______________________
|   pageid! = header?.filecontents pn?
|______________________________________
```

Insert a page id into a file header.

```
PutPageId ___________________________________________
| header? : Header
| pn?     : PageNum
| pageid? : PageId
| header! : Header
| now : Time
|______________________
| heeder!.owner   = header?.owner
| header!.created = header?.created
| header!.updated = now
| header!.expires = header?.expires
| header!.filecontents =
|        header?.filecontents ⊕ {pn? ↦ pageid?}
|____________________________________________________
```

Change expiry date of a file header.

```
SetHeaderExpiry _____________________________________
| header?  : Header
| expires? : Time
| header!  : Header
| now : Time
|______________________
| header!.owner   = header?.owner
| header!.created = header?.created
| header!.updated = now
| header!.expires = expires?
| header!.filecontents = header?.filecontents
|____________________________________________________
```

4.4 Combining operations

The next step is to form the required service operations by combining the suboperations. Basically this can be done using schema conjunction or sequential composition (as discussed in Chapter 1). As we shall later want to argue about the importance of the sequence in which suboperations are performed, and what happens when an operation fails midway through its execution, we shall choose to use sequential composition for combining suboperations.

To pass parameters between suboperations in a sequence it is convenient to introduce some buffers:

```
HeaderBuf ________________
| header : Header
|_________________________
```

holds the header of the pagefile currently being handled.

```
OldPageIdBuf ______________
| oldpageid : PageId
|_________________________
```

holds the id of an existing page (to be read or destroyed).

```
NewPageIdBuf ______________
| newpageid : PageId
|_________________________
```

holds the id of a newly created block in the Page Store.

The progression of an operation can now be described in terms of the states of the subsystems combined with the states of the newly introduced parameter buffers:

$$cPFS_1 \mathrel{\widehat{=}} HeaderBuf \land OldPageIdBuf \land NewPageIdBuf \land HS \land PS$$

In the following, the effect of the individual suboperations on this combined state is described using hiding and renaming (see Chapter 1, section 5).

Operations on Page Store

$$CreatePage_1 \mathrel{\widehat{=}} \Xi cPFS_1 \backslash \Delta PS \backslash \Delta NewPageIdBuf \land CreatePage_{success}[newpageid'/pageid!]$$

$$ReadPage_1 \mathrel{\widehat{=}} \Xi cPFS_1 \backslash \Delta PS \land ReadPage_{success}[oldpageid/pageid?]$$

$$DestroyPage_1 \mathrel{\widehat{=}} \Xi cPFS_1 \backslash \Delta PS \land DestroyPage_{success}[oldpageid/pageid?]$$

$$\begin{array}{ll} \text{SetPageExpiry}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{PS} \land \\ & \quad \text{SetPageExpiry}_{\text{success}}[\text{oldpageid}/\text{pageid?}] \end{array}$$

Operations on Header Store

$$\begin{array}{ll} \text{CreateHeader}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{HS} \land \\ & \quad \text{CreateHeader}_{\text{success}}[\text{header}/\text{header?}] \end{array}$$

$$\begin{array}{ll} \text{ReadHeader}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{HS} \backslash \Delta\text{HeaderBuf} \land \\ & \quad \text{ReadHeade}_{\text{success}}\text{r}\,[\text{header}'/\text{header!}] \end{array}$$

$$\begin{array}{ll} \text{ReplaceHeader}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{HS} \land \\ & \quad \text{ReplaceHeader}_{\text{success}}[\text{header}/\text{header?}] \end{array}$$

$$\begin{array}{ll} \text{DestroyHeader}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{HS} \land \\ & \quad \text{DestroyHeader}_{\text{success}} \end{array}$$

Auxiliary Operations

$$\begin{array}{ll} \text{MakeHeader}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{HeaderBuf} \land \\ & \quad \text{MakeHeader}[\text{header}'/\text{header!}] \end{array}$$

$$\begin{array}{ll} \text{GetPageId}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{OldPageIdBuf} \land \\ & \quad \text{GetPageId}[\text{header}/\text{header?}, \\ & \qquad\qquad \text{oldpageid}'/\text{pageid!}] \end{array}$$

$$\begin{array}{ll} \text{PutPageId}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{HeaderBuf} \land \\ & \quad \text{PutPageId}[\text{header}/\text{header?}, \\ & \qquad\qquad \text{newpageid}/\text{pageid?}, \\ & \qquad\qquad \text{header}'/\text{header!}] \end{array}$$

$$\begin{array}{ll} \text{SetHeaderExpiry}_1 \mathrel{\widehat{=}} & \Xi\text{cPFS}_1 \backslash \Delta\text{HeaderBuf} \land \\ & \quad \text{SetHeaderExpiry}[\text{header}/\text{header?}, \\ & \qquad\qquad\qquad \text{header}'/\text{header!}] \end{array}$$

Apart from these operations a further two slightly more complicated operations are needed, one which destroys all pages belonging to a file, and one which changes the expiry date of all pages belonging to a file.

In order to construct these we first introduce two new operations which, given a page id, will respectively destroy it or change its expiry date. If the id given is the

NullPageId the operations will have no effect at all.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{NotNullId}} \\
\hline
\quad \mathrm{OldPageIdBuf} \\
\hline
\quad \mathrm{oldpageid} \neq NullPageId \\
\hline
\end{array}$$

$$\mathrm{DestroyPage}_{1A} \mathrel{\widehat{=}} \mathrm{DestroyPage}_1 \ \underline{\mathrm{if}}\ \mathrm{NotNullId}$$

$$\mathrm{SetPageExpiry}_{1A} \mathrel{\widehat{=}} \mathrm{SetPageExpiry}_1 \ \underline{\mathrm{if}}\ \mathrm{NotNullId}$$

(For the definition of the if conditional construct see Chapter 1, section 6.1.)

The two required operations can now be defined as:

$$\begin{aligned}
\mathrm{DestroyPages}_1 \mathrel{\widehat{=}}& \\
&\|_{pn?:\mathrm{PageNum}}(\mathrm{GetPageId}_1 \ ;\ \mathrm{DestroyPage}_{1A})
\end{aligned}$$

$$\begin{aligned}
\mathrm{SetPagesExpiry}_1 \mathrel{\widehat{=}}& \\
&\|_{pn?:\mathrm{PageNum}}(\mathrm{GetPageId}_1 \ ;\ \mathrm{SetPageExpiry}_{1A})
\end{aligned}$$

(For the definition of the ‖ interleaving construct see Chapter 1, section 6.3.)

The successful behaviour of the concrete service operations can now be defined by combining the previously defined suboperations in suitable ways.

$$\mathrm{cReadFile}_{success} \mathrel{\widehat{=}} \mathrm{ReadHeader}_1 \ ;\ \mathrm{GetPageId}_1 \ ;\ \mathrm{ReadPage}_1$$

$$\begin{aligned}
\mathrm{cCreateFile}_{success} \mathrel{\widehat{=}}\ & \mathrm{MakeHeader}_1 \ ;\ \mathrm{CreatePage}_1 \ ; \\
& \quad \mathrm{PutPageId}_1 \ ;\ \mathrm{CreateHeader}_1
\end{aligned}$$

$$\begin{aligned}
\mathrm{cDestroyFile}_{success} \mathrel{\widehat{=}}\ & \mathrm{ReadHeader}_1 \ ; \\
& \quad (\mathrm{DestroyHeader}_1 \ \|\ \mathrm{DestroyPages}_1)
\end{aligned}$$

$$\begin{aligned}
\mathrm{cWriteFile}_{success} \mathrel{\widehat{=}}\ & \mathrm{ReadHeader}_1 \ ;\ \mathrm{GetPageId}_1 \ ; \\
& \quad (\mathrm{DestroyPage}_{1A} \ \| \\
& \quad\ (\mathrm{CreatePage}_1 \ ;\ \mathrm{PutPageId}_1 \ ; \\
& \quad\ \ \mathrm{ReplaceHeader}_1))
\end{aligned}$$

$$\mathit{cSetFileExpiry}_{success} \mathrel{\widehat{=}} \mathit{ReadHeader}_1 \mathbin{;} \mathit{SetHeaderExpiry}_1 \mathbin{;} (\mathit{ReplaceHeader}_1 \parallel \mathit{SetPagesExpiry}_1)$$

At this stage the order in which certain of the suboperations are performed is immaterial. In these cases this has been marked by using the $\parallel$ operator rather than the $;$ when combining these to indicate that the order may be reversed if desired.

The Scavenge operation of the PageFile Service need not be implemented since both the Page Store and the Header Store will independently scavenge the appropriate implementation data.

5 Error handling

According to the user's view, the service must be capable of detecting and reporting a number of different types of errors. These are:

NoSuchFile –	occurs if an attempt is made to read, update, destroy or change expiry date for a file which does not exist. In the implementation this corresponds to an attempt to read a non-existing header in a sub-operation. Since all concrete operations manipulating existing files start by reading the fileheader, it will be sufficient if this sub-operation is capable of detecting the error (provided that the subsequent sub-operations are not carried out in this case).
NoSpace –	occurs if an attempt to create a file or to add a page to a file fails due to lack of storage space. In the concrete model of the service this corresponds to failure to create a new page or failure to create a new header.
NoSuchPage –	occurs if an attempt is made to read a non-existent page in an existing file. In the implementation this corresponds to finding the *NullPageId* rather than a specific page id in the appropriate position of the header.
NotOwner –	occurs when an attempt is made to write to, destroy or change the expiry date of an existing file by somebody other that the owner of the file. In the concrete representation this can be detected by checking the owner field of the corresponding header.

In the following the operations of the subsystems will be redefined, to allow for the first two types of errors. Additionally two new anxiliary operations are introduced to cope with the last two types of errors. We shall adopt the convention that all operations on subsystems will return a report indicating whether the operations were successful or not.

The successful report can be described by previously defined Success schema:

```
Success ________________________
|   report! : Report
|__________________________
|   report! = SuccessReport
|__________________________________
```

For the moment, it is assumed that the total operations can be defined by the idealised ones presented in the following sections, which include the error handling just described.

5.1 Page Store subsystem redefined

The only Page Store operation which can occasionally fail is the create page operation, which may return an error report in case the Page Store is full. The capacity of the store is not modelled here, rather we shall let this be a nondeterministic attribute of the underlying implementation.

```
NoPageSpace ____________________________
|   ΞPS
|   nopagespace : Boolean
|   report!     : Report
|_________________________________
|   nopagespace = True
|   report! = NoPageSpaceReport
|_______________________________________
```

$$\mathrm{CreatePage}_{ideal} \mathrel{\widehat{=}} (\mathrm{CreatePage}_{success} \wedge \mathrm{Success}) \oplus \mathrm{NoPageSpace}$$

The remaining operations will always be successful.

$$\mathrm{ReadPage}_{ideal} \mathrel{\widehat{=}} \mathrm{ReadPage}_{success} \wedge \mathrm{Success}$$

$$\mathrm{DestroyPage}_{ideal} \mathrel{\widehat{=}} \mathrm{DestroyPage}_{success} \wedge \mathrm{Success}$$

$$\mathrm{SetPageExpiry}_{ideal} \mathrel{\widehat{=}} \mathrm{SetPageExpiry}_{success} \wedge \mathrm{Success}$$

5.2 Header Store subsystem redefined

As with the create page operation, the create header operation will fail if the header store is full. Again, we shall not model the capacity here but leave this a nondeterministic attribute of the underlying implementation.

```
NoHeaderSpace ________________________________
|   ΞHS
|   noheaderspace : Boolean
|   report! : Report
|________________________
|   noheaderspace = True
|   report! = NoHeaderSpaceReport
|______________________________________________
```

$$\mathrm{CreateHeader}_{ideal} \mathrel{\widehat{=}} (\mathrm{CreateHeader}_{success} \land \mathrm{Success}) \oplus \mathrm{NoHeaderSpace}$$

The readheader operation may fail in case an attempt is made to read a non-existing header (corresponding to an attempt to access a non-existing file on the abstract level).

```
NoSuchHeader _________________________________
|   ΞHS
|   id? : Id
|   report! : Report
|________________________
|   id? ∉ dom headers
|   report! = NoSuchHeaderReport
|______________________________________________
```

$$\mathrm{ReadHeader}_{ideal} \mathrel{\widehat{=}} (\mathrm{ReadHeader}_{success} \wedge \mathrm{Success}) \oplus \mathrm{NoSuchHeader}$$

The remaining two operation will always be successful.

$$\mathrm{ReplaceHeader}_{ideal} \mathrel{\widehat{=}} \mathrm{ReplaceHeader}_{success} \wedge \mathrm{Success}$$

$$\mathrm{DestroyHeader}_{ideal} \mathrel{\widehat{=}} \mathrm{DestroyHeader}_{success} \wedge \mathrm{Success}$$

5.3 Auxiliary errors

As mentioned earlier, two new auxiliary operations will be required.

Check that the current client is owner of the file whose header is held in the header buffer.

$$\mathrm{CheckOwner} \mathrel{\widehat{=}} \mathrm{Success} \oplus \mathrm{NotOwnerError}$$

where

```
NotOwnerError ______________________________
|   header? : Header
|   report! : Report
|   clientnum : UserNum
|_______________
|   header?.owner ≠ clientnum
|   report! = NotOwnerReport
|___________________________________________
```

Check that the page id held in the old page id buffer is a genuine page id rather than the *NullId*.

$$\mathrm{CheckPageId} \;\widehat{=}\; \mathrm{Success} \oplus \mathrm{NoSuchPageError}$$

where

```
NoSuchPageError ________________________
| pageid? : PageId
| report! : Report
|____________________
| pageid? = NullPageId
| report! = NoSuchPageReport
|________________________________________
```

The effect of these new auxiliary operations on the combined system state cPFS_1 can be defined as:

$$\mathrm{CheckOwner}_1 \;\widehat{=}\; \Xi\mathrm{cPFS}_1 \wedge \mathrm{CheckOwner}[\mathrm{header}/\mathrm{header?}]$$

$$\mathrm{CheckPageId}_1 \;\widehat{=}\; \Xi\mathrm{cPFS}_1 \wedge \mathrm{CheckPageId}[\mathrm{oldpageid}/\mathrm{pageid?}]$$

5.4 Combining operations

As with other suboperation parameters the report parameter will be passed on to the subsequent suboperations via a parameter buffer.

```
ErrorState ______________________
| errorstate : Report
|________________________________
```

The contents of this buffer will at a given point in time indicate whether any of the previously executed suboperations failed during the execution of the current service operation. If one suboperation in a sequence for some reason fails, it is often not meaningful to execute the subsequent suboperations. This can be accomplished by specifying that any suboperation should leave the system state unchanged if the error state when the operation is invoked is not *SuccessReport*, i.e. overriding each suboperation specification with:

$$
\begin{array}{|l}
\text{Error} \\
\hline
\quad \Xi \mathrm{ErrorState} \\
\quad \Xi \mathrm{cPFS}_1 \\
\hline
\quad \mathrm{errorstate} \neq \mathit{SuccessReport} \\
\hline
\end{array}
$$

As a last action a service operation should communicate the result of the operation to the user. This basically consists of translating the content of the error buffer into a report type which is known to the user.

$$
\begin{array}{|l}
\text{MakeReport} \\
\hline
\quad \Xi \mathrm{cPFS}_1 \\
\quad \Xi \mathrm{ErrorState} \\
\quad \mathrm{report!} : \mathrm{Report} \\
\hline
\quad \mathrm{errorstate} = \mathit{NoHeaderSpaceReport} \Rightarrow \\
\qquad \mathrm{report!} = \mathit{NoSpaceReport} \\
\quad \mathrm{errorstate} = \mathit{NoPageSpaceReport} \Rightarrow \\
\qquad \mathrm{report!} = \mathit{NoSpaceReport} \\
\quad \mathrm{errorstate} = \mathit{NoSuchHeaderReport} \Rightarrow \\
\qquad \mathrm{report!} = \mathit{NoSuchFileReport} \\
\quad \mathrm{errorstate} = \mathit{NoSuchPageReport} \Rightarrow \\
\qquad \mathrm{report!} = \mathit{NoSuchPageReport} \\
\quad \mathrm{errorstate} = \mathit{NotOwnerReport} \Rightarrow \\
\qquad \mathrm{report!} = \mathit{NotOwnerReport} \\
\hline
\end{array}
$$

The progression of a service operation can be described in terms of the state of the error buffer combined with the system state cPFS_1, defined in the previous section.

$$\mathrm{cPFS}_2 \mathrel{\widehat{=}} \mathrm{ErrorState} \wedge \mathrm{cPFS}_1$$

The individual suboperations affect this combined state as follows.

Operations on Page Storage Subsystem

$\text{CreatePage}_Z \mathrel{\widehat{=}} \text{CreatePage}_{Ideal}[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

$\text{ReadPage}_Z \mathrel{\widehat{=}} \text{ReadPage}_{Ideal}[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

$\text{DestroyPage}_Z \mathrel{\widehat{=}} \text{DestroyPage}_{Ideal}[\text{errorstete}'/\text{report}!] \oplus \text{Error}$

$\text{SetPageExpiry}_Z \mathrel{\widehat{=}} \text{SetPageExpiry}_{Ideal}[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

Operations on Header Storage Subsystem

$\text{CreateHeader}_Z \mathrel{\widehat{=}} \text{CreateHeader}_{Ideal}[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

$\text{ReadHeader}_Z \mathrel{\widehat{=}} \text{ReadHeader}_{Ideal}[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

$\text{ReplaceHeader}_Z \mathrel{\widehat{=}} \text{ReplaceHeader}_{Ideal}[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

$\text{DestroyHeader}_Z \mathrel{\widehat{=}} \text{DestroyHeader}_{Ideal}[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

Auxiliary Operations

$\text{MakeHeader}_Z \mathrel{\widehat{=}} \text{MakeHeader}_1 \oplus \text{Error}$

$\text{GetPageId}_Z \mathrel{\widehat{=}} \text{GetPageId}_1 \oplus \text{Error}$

$\text{PutPageId}_Z \mathrel{\widehat{=}} \text{PutPageId}_1 \oplus \text{Error}$

$\text{SetHeaderExpiry}_Z \mathrel{\widehat{=}} \text{SetHeaderExpiry}_1 \oplus \text{Error}$

$\text{CheckOwner}_Z \mathrel{\widehat{=}} \text{CheckOwner}_1[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

$\text{CheckPageId}_Z \mathrel{\widehat{=}} \text{CheckPageId}_1[\text{errorstate}'/\text{report}!] \oplus \text{Error}$

The special operations to destroy all pages or to change the expiry date for all pages of a file need to be rewritten in terms of the new suboperations, but generally behave in almost exactly the same way as before.

$$\begin{aligned}&\mathrm{DestroyPage}_{2A} \ \widehat{=}\\ &\qquad \mathrm{DestroyPage}_{2}\ \underline{if}\ \mathrm{NotNullPageId}\end{aligned}$$

$$\begin{aligned}&\mathrm{SetPageExpiry}_{2A} \ \widehat{=}\\ &\qquad \mathrm{SetPageExpiry}_{2}\ \underline{if}\ \mathrm{NotNullPageId}\end{aligned}$$

$$\begin{aligned}&\mathrm{DestroyPages}_{2} \ \widehat{=}\\ &\qquad \parallel_{pn?:\mathrm{PageNum}}(\mathrm{GetPageId}_{2} \fatsemi \mathrm{DestroyPage}_{2A})\end{aligned}$$

$$\begin{aligned}&\mathrm{SetPagesExpiry}_{2} \ \widehat{=}\\ &\qquad \parallel_{pn?:\mathrm{PageNum}}(\mathrm{GetPageId}_{2} \fatsemi \mathrm{SetPageExpiry}_{2A})\end{aligned}$$

The concrete service operations can now be redefined in terms of the newly defined suboperations.

$$\begin{aligned}\mathrm{cCreateFile}_{ideal} \ \widehat{=}\ & \mathrm{Success} \fatsemi \mathrm{MakeHeader}_{2} \fatsemi \mathrm{CreatePage}_{2} \fatsemi\\ & \mathrm{PutPageId}_{2} \fatsemi \mathrm{CreateHeader}_{2} \fatsemi\\ & \mathrm{MakeReport}\end{aligned}$$

$$\begin{aligned}\mathrm{cWriteFile}_{ideal} \ \widehat{=}\ & \mathrm{Success} \fatsemi \mathrm{ReadHeader}_{2} \fatsemi \mathrm{CheckOwner}_{2} \fatsemi\\ & \mathrm{GetPageId}_{2} \fatsemi \mathrm{CreatePage}_{2} \fatsemi\\ & (\mathrm{DestroyPage}_{2A} \parallel\\ & \ (\mathrm{PutPageId}_{2} \fatsemi \mathrm{ReplaceHeader}_{2})) \fatsemi\\ & \mathrm{MakeReport}\end{aligned}$$

$$\begin{aligned}\mathrm{cReadFile}_{ideal} \ \widehat{=}\ & \mathrm{Success} \fatsemi \mathrm{ReadHeader}_{2} \fatsemi \mathrm{GetPageId}_{2} \fatsemi\\ & \mathrm{CheckPageId}_{2} \fatsemi \mathrm{ReadPage}_{2} \fatsemi\\ & \mathrm{MakeReport}\end{aligned}$$

$$\begin{aligned}\mathrm{cDestroyFile}_{ideal} \ \widehat{=}\ & \mathrm{Success} \fatsemi \mathrm{ReadHeader}_{2} \fatsemi \mathrm{CheckOwner}_{2} \fatsemi\\ & (\mathrm{DestroyHeader}_{2} \parallel \mathrm{DestroyPages}_{2}) \fatsemi\\ & \mathrm{MakeReport}\end{aligned}$$

$$\begin{aligned}\mathrm{cSetFileExpiry}_{ideal} \ \widehat{=}\ & \mathrm{Success} \fatsemi \mathrm{ReadHeader}_{2} \fatsemi \mathrm{CheckOwner}_{2} \fatsemi\\ & \mathrm{SetHeaderExpiry}_{2} \fatsemi\\ & (\mathrm{ReplaceHeader}_{2} \parallel \mathrm{SetPagesExpiry}_{2}) \fatsemi\\ & \mathrm{MakeReport}\end{aligned}$$

When constructing a service operation by combining suboperations as above, it is important that the concrete state of the system is consistent and corresponds to a well defined abstract state at any point where a suboperation may "fail" (i.e. not return a *SuccessReport*) and thereby in effect abort the remaining suboperations in the sequence. Also, this abstract state must correspond to what the user expects. In most cases this means that the abstract service state must be unchanged whenever a suboperation may "fail".

For the last four service operations above this poses no problems at all, since the preconditions for the entire operations can be (and are) checked before any updating of concrete service state takes place.

The `Create` operation is not quite so simple, as the `CreateHeader` suboperation might fail if the Header Store is full. At this point, however, a new page would have been created. This does not affect the abstract view of the service, but it does violate the compactness constraint stated in section 3. To overcome the problem we might choose to delete the newly created page.

$$
\begin{aligned}
&\mathtt{cCreateFile}_{\mathtt{attempt}} \ \widehat{=} \\
&\qquad \mathtt{Success} \fatsemi \mathtt{MakeHeader}_2 \fatsemi \mathtt{CreatePage}_2 \fatsemi \\
&\qquad\quad \mathtt{PutPageId}_2 \fatsemi \mathtt{CreateHeader}_2 \fatsemi \\
&\qquad\quad \mathtt{MakeReport} \fatsemi \\
&\qquad\quad ((\mathtt{DestroyPage[newpageid/pageid?]} \wedge \\
&\qquad\qquad \Xi\mathtt{cPFS}_1 \setminus \Delta\mathtt{PS})\ \underline{\mathit{if}}\ \mathtt{CreateHeaderError})
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathtt{CreateHeaderError} \ \widehat{=} \\
&\qquad \mathtt{ErrorState} \mid \mathtt{errorstate} = \mathit{NoHeaderSpaceError}
\end{aligned}
$$

6 Implementing one service in terms of another

If the subsystems presented in the previous sections are to be implemented using other services (which of course may reside on different host computers from the PageFile Service), the abstract specification of the subsystem operations must be extended to be able to mirror the types of errors which may be caused directly or indirectly by the use of such services.

According to the Common Service Framework [5] any service operation may at any time return *ServiceErrorReport* as a result of an operation. This is basically to allow for errors in both underlying hardware and software. Note that an operation which return a *ServiceErrorReport* is required to leave the abstract state of the service unchanged.

Also, the communications network which connects the services may fail during the operation. This causes special problems since no indication as to the result of the operation need be given to the requesting service. We will assume that the network layer of the implementation provides full error-recovery and that seen from a user point of view, network errors will never happen.

In the following the operations on the two subservices are first redefined to allow for service errors as described above, and afterwards the impact of these changes on the construction of the service operation implementations presented so far is studied in detail.

6.1 Errors in Page Store

In the Page Store subsystem the following error may occur at any time:

```
PageServiceError ____________________
|   ΞPS
|   report! : Report
|____________________
|   report! = ServiceErrorReport
|____________________________________
```

The complete operations on the subsystem can therefore be defined as:

$$\begin{array}{llll}
CreatePage & \widehat{=} & CreatePage_{ideal} & \lor\ PageServiceError \\
ReadPage & \widehat{=} & ReadPage_{ideal} & \lor\ PageServiceError \\
DestroyPage & \widehat{=} & DestroyPage_{ideal} & \lor\ PageServiceError \\
SetPageExpiry & \widehat{=} & SetPageExpiry_{ideal} & \lor\ PageServiceError
\end{array}$$

6.2 Errors in Header Store

In the Header Store subsystem the equivalent errors may occur at any time:

$$\begin{array}{|l}
\multicolumn{1}{l}{HeaderServiceError} \\
\hline
\quad \Xi HS \\
\quad report! : Report \\
\hline
\quad report! = \mathit{ServiceErrorReport} \\
\hline
\end{array}$$

The total operations on the subsystem can now be defined as:

$$\begin{array}{llll}
CreateHeader & \widehat{=} & CreateHeader_{ideal} & \lor\ HeaderServiceError \\
ReadHeader & \widehat{=} & ReadHeadar_{ideal} & \lor\ HeaderServiceError \\
ReplaceHeader & \widehat{=} & ReplaceHeader_{ideal} & \lor\ HeaderServiceError \\
DestroyHeader & \widehat{=} & DestroyHeader_{ideal} & \lor\ HeaderServiceError
\end{array}$$

6.3 Constructing the service operations

We shall now see what changes will be required to the service operations in order to handle the newly introduced error types.

The first change is the obvious one of changing the MakeReport schema, so that the new errors may be reported to the user of the service. The fact that the PageFile Service makes use of other services is transparent to the user. Errors occurring in such services or during communications with such services, should therefore be reported as if they occurred in the PageFile Service itself.

cMakeReport

$\Xi cPFS_1$
$\Xi ErrorState$
$report! : Report$

$errorstate = NoHeaderSpaceReport \Rightarrow$
$\quad report! = NoSpaceReport$
$errorstate = NoPageSpaceReport \Rightarrow$
$\quad report! = NoSpaceReport$
$errorstate = NoSuchHeaderReport \Rightarrow$
$\quad report! = NoSuchFileReport$
$errorstate = NoSuchPageReport \Rightarrow$
$\quad report! = NoSuchPageReport$
$errorstate = NotOwnerReport \Rightarrow$
$\quad report! = NotOwnerReport$
$errorstate = ServiceErrorReport \Rightarrow$
$\quad report! = ServiceErrorReport$

We shall reconsider each of the specifications of concrete pagefile operations presented in the previous section, taking into consideration that any operation on a subsystem may fail, and that correct sequencing of suboperations therefore is even more crucial than before in order to ensure that the system state is always consistent.

The ReadFile operation does not change the system state at any point and can therefore cause no problems.

$$cReadFile \mathrel{\widehat{=}} Success \fatsemi ReadHeader_2 \fatsemi GetPageId_2 \fatsemi CheckPageId_2 \fatsemi ReadPage_2 \fatsemi cMakeReport$$

The CreateFile operation takes the form:

$$cCreateFile \mathrel{\widehat{=}} Success \fatsemi MakeHeader_2 \fatsemi CreatePage_2 \fatsemi PutPageId_2 \fatsemi CreateHeader_2 \fatsemi cMakeReport$$

CreatePage may fail during the CreateHeader operation. In this case a new page would have been created, which was not referenced from any file. This violates the compactness constraint. We could of course try to repeat the failed suboperation, or attempt to delete the just created page, but there is no guarantee that we will succeed in doing so within a reasonable time. We therefore have two choices: either to suspend operation indefinitely (i.e. in effect closing down the service) or to ease the compactness constraint and just state that we will attempt to obtain compactness.

The destroy operation takes the form:

$$\begin{aligned}\mathrm{cDestroyFile}_{attempt} \mathrel{\widehat{=}}\ & \mathrm{Success} \fatsemi \mathrm{ReadHeader}_Z \fatsemi \mathrm{CheckOwner}_Z \fatsemi \\ & (\mathrm{DestroyHeader}_Z \parallel \mathrm{DestroyPages}_Z) \fatsemi \\ & \mathrm{cMakeReport}\end{aligned}$$

Here, if the pages are destroyed first followed by the header and the latter operation fails, we will end up with a file header referring to pages which do not exist any longer (and thus violates the specification). If the header is destroyed first and then the pages, and part of the latter operation fails, we will end up with some pages which no longer correspond to an existing header and thus violate the compactness constraint as above. In reality what we can do by ignoring the constraint is to make the outcome of the operation independent on the outcome of DestroyPages and we can therefore create the report before this suboperation is performed.

$$\begin{aligned}\mathrm{cDestroyFile} \mathrel{\widehat{=}}\ & \mathrm{Success} \fatsemi \mathrm{ReadHeader}_Z \fatsemi \mathrm{CheckOwner}_Z \fatsemi \\ & \mathrm{DestroyHeader}_Z \fatsemi \mathrm{cMakeReport} \fatsemi \\ & \mathrm{DestroyPages}_Z\end{aligned}$$

The write operation takes the form:

$$\begin{aligned}\mathrm{cWriteFile}_{attempt} \mathrel{\widehat{=}}\ & \mathrm{Success} \fatsemi \mathrm{ReadHeader}_Z \fatsemi \mathrm{CheckOwner}_Z \fatsemi \\ & \mathrm{GetPagaId}_Z \fatsemi \mathrm{CreatePage}_Z \fatsemi \\ & (\mathrm{DestroyPage}_{ZA} \parallel \\ & \quad (\mathrm{PutPageId}_Z \fatsemi \mathrm{ReplaceHeader}_Z)) \fatsemi \\ & \mathrm{MakeReport}\end{aligned}$$

Here we observe the same problem with the compactness constraint as before. If DestroyPage is performed before ReplaceHeader and replace header fails, the old page has been corrupted whereas the new one has not been completely created yet. If DestroyPage is performed after ReplaceHeader and fails we have again violated the compactness constraint. Removing the constraint makes the result of the operation independent of DestroyPage.

$$\begin{aligned} cWriteFile \;\hat{=}\;\; & Success \fatsemi ReadHeader_2 \fatsemi CheckOwner_2 \fatsemi \\ & GetPageId_2 \fatsemi CreatePage_2 \fatsemi PutBlockId_2 \fatsemi \\ & ReplaceHeader_2 \fatsemi cMakeReport \fatsemi \\ & DestroyPage_{2A} \end{aligned}$$

Note that we could use CreateHeader and DestroyHeader instead of the atomic ReplaceHeader:

$$\ldots\; CreatePage \fatsemi CreateHeader \fatsemi DestroyHeader \fatsemi DestroyPage \;\ldots$$

This however presents us with another problem; if DestroyHeader fails we end up having both the old and the new file in the system, which certainly violates the specification of WriteFile.

The SetFileExpiry operation takes the form:

$$\begin{aligned} & cSetFileExpiry_{attempt} \;\hat{=} \\ & \quad Success \fatsemi ReadHeader_2 \fatsemi \\ & \quad\; CheckOwner_2 \fatsemi SetHeaderExpiry_2 \fatsemi \\ & \quad\; (ReplaceHeader \parallel SetPagesExpiry_2) \fatsemi \\ & \quad\; cMakeReport \end{aligned}$$

No matter whether we perform the SetPagesExpiry operation before or after the SetHeaderExpiry operation, if the later operation fails, the expiry time of all pages will not be the same as that of the header and we have violated the expiry constraint. Obviously we have to ignore this constraint.

However this is not enough. The state invariant specifies that the pages belonging to a header must not expire before the header. This means that if the lifetime of a file is to be increased, then SetPagesExpiry must be performed before ReplaceHeader, so that the state will be consistent should the latter operation fail. However, if the lifetime is to be decreased then the operations should be performed in the reverse order.

```
IncreaseLifetime ______________________
|   HeaderBuf
|   expires? : Time
|_____________________
|
|   expires? ≥ header.expires
|______________________________________
```

```
cSetFileExpiry  ≙
                  Success ⨾ ReadHeader2 ⨾
                   CheckOwner2 ⨾ SetHeaderExpiry2
                   ((SetPagesExpiry2 ⨾ ReplaceHeader2)
                        if IncreaseLifetime else
                    (ReplaceHeader2 ⨾ SetPagesExpiry2)) ⨾
                   cMakeReport
```

6.4 Expiry during operations

The specification we now have derived is quite convincing. There is however still one outstanding problem which is perhaps not obvious; what happens if a file expires while it is being read or updated? If we were unfortunate, the components of that file might be scavenged by the subsystems before the operation was completed. In the specification above this might result in an attempt to manipulate an expired entity which would result in a *ServiceErrorReport*. This is perhaps not what one might expect but it does fulfil the requirements since any service is always allowed to return *ServiceErrorReport* as a result.

The other obvious way of coping with this problem is to make sure that it does not happen. If we assume that the maximum duration of any operation is *DeltaTime*, we could get around the problem by specifying that the expiry time of the subcomponents should be *DeltaTime* after the expiry time of the file, and at the same time change the abstract specification such that no attempt will be made to access a file after it has expired. To do this the NoSuchFile schema from section 2.5 should be substituted by:

```
NoSuchFile ______________________________
|    ≡PFS
|    id? : Id
|____________________
|    id? ∉ dom files ∨
|    (files id?).expires < now
|    report! = NoSuchFileReport
|________________________________________
```

However, as the frequency with which this type of error can occur is negligible we shall regard service error as an acceptable result under these circumstances.

7 Implementations of subsystems

In this section we shall present possible implementations of the two subsystems which were originally defined in section 3 and extended in the subsequent sections.

Both the Header Store and the Page Store will be implemented using the Block Storage Service [6], with state defined by the schema SS. Seen from the Block Storage Service the owner of the blocks used to represent pagefiles is the PageFile Service itself. In order to distinguish blocks which are used to represent pages from blocks used to represent headers, we shall use the first data byte of each block to indicate to which of the subservices the block belongs.

Two constants are used to identify the block type

```
│ PageBlock   : Byte
│ HeaderBlock : Byte
├──────────────────────
│ PageBlock ≠ HeaderBlock
```

Any block belonging to the PageFile Service will be marked as belonging to either the Page Store subsystem or the Header Store subsystem.

```
MarkedBlocks ─────────────────────────────────
│ SS
├──────────────
│ ∀ b:ran blocks •
│   b.owner = PageFileService ⇒
│     b.data(1) ∈ {PageBlock, HeaderBlock}
└─────────────────────────────────────────────
```

An implementation making use of a subsystem is obliged only to use the subsystem within its defined scope. If this cannot be guaranteed (i.e. if the implementation cannot be proven correct with respect to the specification of the subsystem), there is an awkward problem. The implementor can either ignore the problem (at some risk) or can perform simple consistency checks and at least return some kind of error reports if obvious inconsistencies occur, thus making the debugging of user programs somewhat easier.

In the following we shall regard obvious inconsistencies as being equivalent to service errors and treat them as such.

7.1 Implementation of the Page Store

Pages and blocks are both defined as sequences of bytes. Provided that the page size is less that the block size (to allow for the byte indicating the block type), it is therefore a trivial matter to represent a page in terms of a block. We shall choose to let each page be represented by a block and shall let the page be identified by the name of the block representing it.

```
PageId  ≙  BlockId
```

Two straightforward operations describe the conversion from pages to blocks and vice-versa.

```
| PackPage    : Page → BlockData
| UnPackPage  : BlockData ⇸ Page
|--------------------------------
| ∀ p:Page •
|    (PackPage p) for PageSize+1 =
|       PageBlock ⁀ p
|
| ∀ b:BlockData | b(1) = PageBlock •
|    UnPackPage b =
|       b after 1 for PageSize
```

The representation relation for the page store subsystem can be defined simply as:

```
RelPS ________________________________
|  PS
|  SS
|-------------
|  ∀ pi:dom pages •
|       b.owner = PageFileService
|       b.expires = p.expires
|       b.data = PackPage p.contents
|    where
|       b ≙ (blocks pi)
|       p ≙ (pages pi)
|  newpageids = newids
|_____________________________________
```

It should now be fairly obvious that each page store operation can be implemented in terms of exactly one corresponding block service operation, together with some data conversion. Since the implementation of the operations is so straightforward we shall use simple schema conjunction (see Chapter 1) in defining the concrete operations. In the following a number of concrete operations will be presented, each corresponding to one of the earlier defined abstract subsystem operations.

Create a new page:

```
cCreatePage ____________________________________________
|   Create[blockdata/data?, expires?/expiry?,
|          pageid!/id!, report/report!]
|   page?     : Page
|   expires?  : Time
|   pageid!   : PageId
|   report!   : Report
|   blockdata : BlockData
|   report    : Report
|__________________
|   blockdata = PackPage page?
|   report = SuccessReport ⇒
|        report! = SuccessReport
|   report = NoSpaceReport ⇒
|        report! = NoPageSpaceReport
|   report ∉ {SuccessReport, NoSpaceReport} ⇒
|        report! = ServiceErrorReport
|________________________________________________________
```

Note, that if the block storage operation returns with an unexpected error, a *ServiceError* report will be returned.

Read a Page:

```
cReadPage ___________________________________________
| Read[pageid?/id?, blockdata/data!,
|      report/report!]
| pageid?   : PageId
| page!     : Page
| report!   : Report
| blockdata : BlockData
| report    : Report
|_____________________________________
| page! = UnPackPage blockdata
| report = SuccessReport ⇒
|      blockdata(1) = PageBlock ⇒
|           report! = SuccessReport
|      blockdata(1) ≠ PageBlock ⇒
|           report! = ServiceErrorReport
| report ≠ SuccessReport ⇒
|      report! = ServiceErrorReport
|____________________________________________________
```

If the block does not exist or it is not a page block, the specification of the operation has been violated, and a service error is reported.

Remove a page:

```
cDestroyPage ________________________________________
| Destroy[pageid?/id?, report/report!]
| pageid? : PageId
| report! : Report
| report  : Report
|_____________________________________
| report = SuccessReport ⇒
|      report! = SuccessReport
| report ≠ SuccessReport ⇒
|      report! = ServiceErrorReport
|____________________________________________________
```

```
cSetPageExpiry
  SetExpiry[pageid?/id?, expires?/expiry?,
            report/report!]
  pageid?  : PageId
  expires? : Time
  report!  : Report
  report   : Report
  ─────────────────────────────────────
  report = SuccessReport ⇒
       report! = SuccessReport
  report ≠ SuccessReport ⇒
       report! = ServiceErrorReport
```

7.2 Implementation of the Header Store

Assuming that a header can be represented as a fixed-length sequence of bytes:

$$\mathrm{HeaderRep} \mathrel{\widehat{=}} 0 \,..\, HeaderRepSize - 1 \rightarrow \mathrm{Byte}$$

and assuming that

$$HeaderRepSize < BlockSize$$

the concrete representation of the headers can be defined in much the same way as the representation of the pages.

Of course, the disadvantage of this representation is that the maximum allowed number of pages in a pagefile, $MaxPages$, would be fairly small. For an attempt at a more advanced and flexible implementation of a header store see [11].

In the following we shall assume the existence of a set of operations to convert to and from this representation.

```
  HeaderToRep : Header ↣ HeaderRep
  RepToHeader : HeaderRep ⤖ Header
  ──────────────────────────────
  RepToHeader = HeaderToRep⁻¹
```

Given the conversion functions it is a trivial matter to represent a header in terms of a block and a header can be identified in terms of the block by which it is represented.

```
Id ≙ BlockId
```

The conversion functions could be defined as:

```
│  PackHeader   : Header → BlockData
│  UnPackHeader : BlockData ⇸ Header
├──────────────────────────────────────────────
│  ∀ h:Header •
│     (PackHeader h) for HeaderRepSize =
│        HeaderBlock⁀(HeaderToRep h)
│
│  ∀ b:BlockData | b.data(1) = HeaderBlock •
│     UnPackHeader b =
│       RepToHeader (b after 1 for HeaderRepSize)
│
```

The representation relation for the Header Store is very much like that for Page Store.

```
 RelHS ────────────────────────────────────────┐
│  HS
│  SS
├──────────────────
│  ∀ h:dom headers •
│       hd.owner = PageFileService
│       hd.expires = b.expires
│       b.deta = PackHeader hd
│    where
│       hd ≙ (headers h)
│       b  ≙ (blocks h)
│
│  newheaderids = newids
└──────────────────────────────────────────────┘
```

The concrete operations on the Header Store resemble very much the corresponding operations on the Page Store.

Create a new header:

```
cCreateHeader
    Create[blockdata/data?, expires?/expiry,
           id!, report/report!]
    header?   : Header
    expires?  : Time
    id!       : Id
    report!   : Report
    blockdata : BlockData
    report    : Report
    ─────────────────────────────────────
    blockdata = PackHeader header?
    report = SuccessReport ⇒
         report! = SuccessReport
    report = NoSpaceReport ⇒
         report! = NoHeaderSpaceReport
    report ∉ {SuccessReport, NoSpaceReport} ⇒
         report! = ServiceErrorReport
```

Read a header:

```
cReadHeader ____________________________________________
| Read[id?, blockdata/data!, report/report!]
| id?       : Id
| header!   : Header
| report!   : Report
| blockdata : BlockData
| report    : Report
|_______________________________
| header! = UnPackHeader blockdata
| report = SuccessReport ⇒
|     blockdata(1) = HeaderBlock ⇒
|         report! = SuccessReport
|     blockdata(1) ≠ PageBlock ⇒
|         report! = NoSuchHeaderReport
| report ∈ {NoSuchBlockReport, NotOwnerReport} ⇒
|     report! = NoSuchHeaderReport
| report ∉ {SuccessReport, NoSuchBlockReport,
|             NotOwnerReport} ⇒
|     report! = ServiceErrorReport
|_______________________________________________________
```

If the requested block does not belong to the PageFile Service or if it is not marked as a header block, it is reported as non-existing.

Remove a header:

```
cDestroyHeader _______________________________
| Destroy[id?, report/report!]
| id?     : Id
| report! : Report
| report  : Report
|_______________________________
| report = SuccessReport ⇒
|     report! = SuccessReport
| report ≠ NetErrorReport ⇒
|     raport! = ServiceErrorReport
|_____________________________________________
```

Replace one header with another:

```
cReplaceHeader
    Replace[id?, blockdata/data?, id!,
            report/report!]
    id?        : Id
    heeder?    : Header
    id!        : Id
    report!    : Report
    blockdata  : BlockData
    report     : Report
    ─────────────────────────────────────────
    blockdata = PackHeader header?
    report = SuccessReport ⇒
        report! = SuccessReport
    report ≠ NetErrorReport ⇒
        report! = ServiceErrorReport
```

This completes the specification of the concrete operations.

8 Conclusion

Despite restructuring the implementation specification a number of times during its development, in an attempt to make it easier to understand, the version presented here is still by no means straightforward to assimilate.

One of the advantages of the schema composition techniques, namely being able to abstract part of a specification under a simple name, is also one of its main disadvantages. In larger specifications, such as this one, it is all too easy to hide the detail so well that it is overlooked!

Here, as in the Block Storage Service Implementor Manual [6], we bave tried to present a design in a form suitable for a programmer rather than for a proof of correctness. The refinement steps demonstrate design decisions, and are probably too large to realistically expect a complete proof to be carried out by hand. In addition, the notation has been kept within the Z framework, although a small number of extra schema operators have been added, in particular to handle iteration and interleaving (see Chapter 1, section 6). For an example of how a Z specification could be further refined towards a programming language, see [12].

The provision of computer based tools may help with refinement in the future. However, even without these tools, the use of a formal notation gives the designer more confidence and understanding of the internal design of the system before the coding stage.

Acknowledgements

Thank you to Carroll Morgan and Tim Gleeson, former members of the Distributed Computing Software project, for providing some of the experiences on which this monograph is based. In addition, Tim Gleeson made invaluable comments on an earlier draft.

Also, thank you to those working on the development of Z in the several related projects at the Programming Research Group who have provided inspiration.

The Distributed Computing Software project has been funded by a grant from the Science and Engineering Research Council.

References

1. Sufrin, B.A. (Editor) "Z Handbook", Draft 1.1, *Programming Research Group, Oxford University*, (1986).

2. Spivey, J.M. "The Z Library - A Reference Manual", *Programming Research Group, Oxford University*, (1986).

3. King, S., Sørensen, I.H., Woodcock, J. "Z: Concrete and Abstract Syntaxes", Version 1.0, *Programming Research Group, Oxford University*, (1987).

4. Hayes, I.J. (Editor) "Specification Case Studies", *Prentice-Hall International Series in Computer Science,* (1987).

5. Bowen, J., Gimson, R.B., Topp-Jørgensen, S. "The Specification of Network Services", Technical Monograph PRG-61, *Programming Research Group, Oxford University*, (1987).

6. Gimson, R.B. "The Formal Documentation of a Block Storage Service", Technical Monograph PRG-62, *Programming Research Group, Oxford University*, (1987).

7. Woodcock, J. "Structuring Specifications - Notes on the Schema Notation", *Programming Research Group, Oxford University*, (1986).

8. Sørensen, I.H. "Structured Programming with Schemas", *Programming Research Group, Oxford University*, (1986).

9. Josephs, M. "Formal Methods for Stepwise Refinement in the Z Specification Language" , *Programming Research Group, Oxford University*, (1986).

10. Wordsworth, J. "A Z Development Method", *IBM, Hursley Park*, UK, (1987).

11. Gimson, R.B. "PageFile Service - Implementor Manual", DCS Project working paper, *Programming Research Group, Oxford University*, (1987).

12. Morgan, C.C. "The Specification Statement: Formal Treatment", Course Notes for Software Engineering, *Programming Research Group, Oxford University*, (1986).

Appendix A

Index of formal definitions

The following index lists the page numbers on which each formal name is defined in the text. In particular, all schema names are included to aid cross reference. Schemas names in the index with a "*" rather than a page number next to them are defined in the Block Storage Service [6]. Names which have a special symbol (Δ, ϕ, $\equiv$, c) as a prefix are listed after the corresponding base name. Note that for a schema S, unless otherwise defined, it is assumed that the following definitions exist if required:

$$\Delta S \ \widehat{=}\ S \wedge S'$$
$$\equiv S \ \widehat{=}\ \Delta S \mid \theta S' = \theta S$$

Appendix B

Glossary of Z notation

A glossary of the Z mathematical and schema notation used in this monograph is included here for easy reference. Readers should note that the definitive concrete and abstract syntax for Z is available elsewhere [3].

Z Reference Glossary

Mathematical Notation

1. Definitions and declarations.

Let x, x_i be identifiers, t, t_i be terms and T, T_i be sets.

$[T_1, T_2, \ldots]$	Introduction of given sets.
$x \mathrel{\widehat{=}} t$	Definition of x as syntactically equivalent to t.
$x ::= x_1 \langle\langle t_1 \rangle\rangle \mid \ldots \mid x_n \langle\langle t_n \rangle\rangle$	Data type definition (the $\langle\langle t \rangle\rangle$ terms are optional).
$x : T$	Declaration of x as type T.
$x_1 : T_1 ; \ldots ; x_n : T_n$	List of declarations.
$x_1, \ldots, x_n : T$	Declarations of the same type: $\mathrel{\widehat{=}} x_1 : T ; \ldots ; x_n : T$.

2. Logic.

Let P, Q be predicates and D declarations.

$\neg P$	Negation: "not P".
$P \wedge Q$	Conjunction: "P and Q".
$P \vee Q$	Disjunction: "P or Q": $\mathrel{\widehat{=}} \neg(\neg P \wedge \neg Q)$.
$P \Rightarrow Q$	Implication: "P implies Q" or "if P then Q": $\mathrel{\widehat{=}} \neg P \vee Q$.
$P \Leftrightarrow Q$	Equivalence: "P is logically equivalent to Q": $\mathrel{\widehat{=}} (P \Rightarrow Q) \wedge (Q \Rightarrow P)$.
true	Logical constant.
false	$\mathrel{\widehat{=}} \neg \mathsf{true}$
$\forall D \bullet P$	Universal quantification: "for all D, P holds".
$\exists D \bullet P$	Existential quantification: "there exists D such that P".
$\exists_1 D \bullet P$	Unique existence: "there exists a unique D such that P".
$\forall D \mid P \bullet Q$	$\mathrel{\widehat{=}} (\forall D \bullet P \Rightarrow Q)$.
$\exists D \mid P \bullet Q$	$\mathrel{\widehat{=}} (\exists D \bullet P \wedge Q)$.
$P \ \underline{\mathsf{where}}\ D \mid Q$	Where clause: $\mathrel{\widehat{=}} \exists D \mid Q \bullet P$
$P \ \underline{\mathsf{where}}\ x_1 \mathrel{\widehat{=}} t_1 ; \ldots ; x_n \mathrel{\widehat{=}} t_n$	Where clause: P holds, with the syntactic definition(s) defined locally.
$D \vdash P$	Theorem: $\mathrel{\widehat{=}} \vdash \forall D \bullet P$.

3. Sets.

Let S, T and X be sets; t, t_i terms; P a predicate and D declarations.

$t_1 = t_2$	Equality between terms.
$t_1 \neq t_2$	Inequality: $\mathrel{\widehat{=}} \neg(t_1 = t_2)$.
$t \in S$	Set membership: "t is an element of S".
$t \notin S$	Non-membership: $\mathrel{\widehat{=}} \neg(t \in S)$.
$\emptyset$	Empty set: $\mathrel{\widehat{=}} \{ x : X \mid \mathsf{false} \}$.
$S \subseteq T$	Set inclusion: $\mathrel{\widehat{=}} (\forall x : S \bullet x \in T)$.
$S \subset T$	Strict set inclusion: $\mathrel{\widehat{=}} S \subseteq T \wedge S \neq T$.
$\{ t_1, t_2, \ldots, t_n \}$	The set containing $t_1, t_2, \ldots$ and t_n.
$\{ D \mid P \bullet t \}$	The set of t's such that given the declarations D, P holds.
$\{ D \mid P \}$	Given $D \mathrel{\widehat{=}} x_1 : T_1 ; \ldots ; x_n : T_n$, $\mathrel{\widehat{=}} \{ D \mid P \bullet (x_1, \ldots, x_n) \}$.
$\{ D \bullet t \}$	$\mathrel{\widehat{=}} \{ D \mid \mathsf{true} \bullet t \}$.
$(t_1, t_2, \ldots, t_n)$	Ordered n-tuple of $t_1, t_2, \ldots$ and t_n.
$T_1 \times T_2 \times \ldots \times T_n$	Cartesian product: the set of all n-tuples such that the i th component is of type T_i.
$\mathbb{P}\, S$	Powerset: the set of all subsets of S.
$\mathbb{P}_1\, S$	Non-empty powerset: $\mathrel{\widehat{=}} \mathbb{P}\, S \setminus \{\emptyset\}$.
$\mathbb{F}\, S$	Set of finite subsets of S: $\mathrel{\widehat{=}} \{ T : \mathbb{P}\, S \mid T \text{ is finite} \}$.
$\mathbb{F}_1\, S$	Non-empty finite set: $\mathrel{\widehat{=}} \mathbb{F}\, S \setminus \{\emptyset\}$.

$S \cap T$	Set intersection: given $S, T: \mathbb{P}\,X$, $\widehat{=} \{x:X \mid x \in S \land x \in T\}$.
$S \cup T$	Set union: given $S, T: \mathbb{P}\,X$, $\widehat{=} \{x:X \mid x \in S \lor x \in T\}$.
$S \setminus T$	Set difference: given $S, T: \mathbb{P}\,X$, $\widehat{=} \{x:X \mid x \in S \land x \notin T\}$.
$\bigcap SS$	Distributed set intersection: given $SS: \mathbb{P}\,(\mathbb{P}\,X)$, $\widehat{=} \{x:X \mid (\forall S:SS \bullet x \in S)\}$.
$\bigcup SS$	Distributed set union: given $SS: \mathbb{P}\,(\mathbb{P}\,X)$, $\widehat{=} \{x:X \mid (\exists S:SS \bullet x \in S)\}$.
$\#S$	Size (number of distinct elements) of a finite set.
$\mu\, D \mid P \bullet t$	Arbitrary choice from the set $\{ D \mid P \bullet t \}$.
$\mu\, D \bullet t$	$\widehat{=} \mu\, D \mid \mathit{true} \bullet t$

4. Relations.

A relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations. Let X, Y, and Z be sets; $x:X$; $y:Y$; and $R:X \leftrightarrow Y$.

$X \leftrightarrow Y$	The set of relations from X to Y: $\widehat{=} \mathbb{P}\,(X \times Y)$.
$x\ R\ y$	x is related by R to y: $\widehat{=} (x, y) \in R$. (R is often underlined for clarity.)
$x \mapsto y$	Maplet: $\widehat{=} (x, y)$.
$\mathrm{dom}\ R$	The domain of a relation: $\widehat{=} \{ x:X \mid \exists y:Y \bullet x\ R\ y \}$.
$\mathrm{ran}\ R$	The range of a relation: $\widehat{=} \{ y:Y \mid \exists x:X \bullet x\ R\ y \}$.
$R_1 \mathbin{;} R_2$	Forward relational composition: given $R_1: X \leftrightarrow Y$; $R_2: Y \leftrightarrow Z$, $\widehat{=} \{ x:X;\ z:Z \mid \exists y:Y \bullet x\ R_1\ y \land y\ R_2\ z \}$.
$R_1 \circ R_2$	Relational composition: $\widehat{=} R_2 \mathbin{;} R_1$.
R^{-1}	Inverse of relation R: $\widehat{=} \{ y:Y;\ x:X \mid x\ R\ y \}$.
$\mathrm{id}\ X$	Identity function on the set X: $\widehat{=} \{ x:X \bullet x \mapsto x \}$.
R^{k}	The relation R composed with itself k times: given $R: X \leftrightarrow X$, $R^0 \widehat{=} \mathrm{id}\ X$, $R^{k+1} \widehat{=} R^{k} \circ R$.
R^{*}	Reflexive transitive closure: $\widehat{=} \bigcup \{ n:\mathbb{N} \bullet R^n \}$.
R^{+}	Non-reflexive transitive closure: $\widehat{=} \bigcup \{ n:\mathbb{N}_1 \bullet R^n \}$.
$R(\!\mid S \mid\!)$	Relational image: given $S: \mathbb{P}\,X$, $\widehat{=} \{ y:Y \mid \exists x:S \bullet x\ R\ y \}$.
$S \lhd R$	Domain restriction to S: given $S: \mathbb{P}\,X$, $\widehat{=} \{x:X;\ y:Y \mid x \in S \land x\ R\ y\}$.
$S \ntriangleleft R$	Domain subtraction: given $S: \mathbb{P}\,X$, $\widehat{=} (X \setminus S) \lhd R$.
$R \rhd T$	Range restriction to T: given $T: \mathbb{P}\,Y$, $\widehat{=} \{x:X;\ y:Y \mid x\ R\ y \land y \in T\}$.
$R \ntriangleright T$	Range subtraction of T: given $T: \mathbb{P}\,Y$, $\widehat{=} R \rhd (Y \setminus T)$.
$\_R\_$	Infix relation declaration (often underlined in use for clarity).

5. Functions.

A function is a relation with the property that for each element in its domain there is a unique element in its range related to it. As functions are relations all the operators for relations also apply to functions.

$X \nrightarrow Y$	The set of partial functions from X to Y: $\widehat{=} \{f : X \leftrightarrow Y \mid \forall x : \mathrm{dom}\ f \bullet (\exists_1 y : Y \bullet x\ f\ y)\}$.
$X \rightarrow Y$	The set of total functions from X to Y: $\widehat{=} \{f : X \nrightarrow Y \mid \mathrm{dom}\ f = X\}$.

$X \pinj Y$	The set of partial injective (one-to-one) functions from X to Y: $\widehat{=} \{f : X \pfun Y \mid \forall y : \mathrm{ran}\ f \bullet (\exists_1 x : X \bullet f\ x = y)\}$.
$X \inj Y$	The set of total injective functions from X to Y: $\widehat{=} (X \pinj Y) \cap (X \fun Y)$.
$X \psurj Y$	The set of partial surjective functions from X to Y: $\widehat{=} \{f : X \pfun Y \mid \mathrm{ran}\ f = Y\}$.
$X \surj Y$	The set of total surjective functions from X to Y: $\widehat{=} (X \psurj Y) \cap (X \fun Y)$.
$X \bij Y$	The set of total bijective (injective and surjective) functions from X to Y: $\widehat{=} (X \surj Y) \cap (X \inj Y)$.
$X \ffun Y$	The set of finite partial functions from X to Y: $\widehat{=} \{f : X \pfun Y \mid f \in \mathbb{F}\,(X \times Y)\}$.
$\pfun\ \pinj\ \psurj\ \pbij$	Partial functions.
$\fun\ \inj\ \surj\ \bij$	Total functions.
$\ffun\ \finj\ \fsurj\ \fbij$	Finite functions.
$f_1 \oplus f_2$	Functional overriding: given $f_1, f_2 : X \pfun Y$, $\widehat{=} (\mathrm{dom}\ f_2 \ndres f_1) \cup f_2$.
$f\ \_$	Prefix function declaration (default if no underlines used).
$(\_\ f\ \_)$	Infix function declaration (often underlined in use for clarity).
$\_\ f$	Postfix function declaration.
$f\ t$	The function f applied to t.
$f(t)$	$\widehat{=} f\ t$.
$\lambda D \mid P \bullet t$	Lambda-abstraction: the function that, given an argument x of type X such that P holds, the result is t. Given $D \widehat{=} x_1 : T_1 ; \ldots ; x_n : T_n$, $\widehat{=} \{D \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$.
$\lambda D \bullet t$	$\widehat{=} \lambda D \mid \mathrm{true} \bullet t$

6. Numbers.

Let m, n be natural numbers.

$\mathbb{N}$	The set of natural numbers (non-negative integers).
$\mathbb{N}_1$	The set of strictly positive natural numbers: $\widehat{=} \mathbb{N} \setminus \{0\}$.
$\mathbb{Z}$	The set of integers (positive, zero and negative).
$\mathrm{succ}\ n$	Successive ascending natural number.
$\mathrm{pred}\ n$	Previous descending natural number: $\widehat{=} \mathrm{succ}^{-1}\ n$.
$m + n$	Addition: $\widehat{=} \mathrm{succ}^n\ m$.
$m - n$	Subtraction: $\widehat{=} \mathrm{pred}^n\ m$.
$m * n$	Multiplication: $\widehat{=} (\_ + m)^n\ 0$.
$m\ \underline{\mathrm{div}}\ n$	Integer division.
$m\ \underline{\mathrm{mod}}\ n$	Modulo arithmetic.
m^n	Exponentiation: $\widehat{=} (\_ * m)^n\ 1$.
$m \leqslant n$	Less than or equal, Ordering: $\_\leqslant\_ \ \widehat{=} \mathrm{succ}^*$.
$m < n$	Less than, Strict ordering: $\widehat{=} m \leqslant n \land m \neq n$.
$m \geqslant n$	Greater than or equal: $\widehat{=} n \leqslant m$.
$m > n$	Greater than: $\widehat{=} n < m$.
$m \mathbin{..} n$	Range: $\widehat{=} \{k : \mathbb{N} \mid m \leqslant k \land k \leqslant n\}$.
$\min S$	Minimum of a finite set; for $S : \mathbb{F}_1\ \mathbb{N}$, $\min S \in S \land (\forall x : S \bullet x \geqslant \min S)$.
$\max S$	Maximum of a finite set; for $S : \mathbb{F}_1\ \mathbb{N}$, $\max S \in S \land (\forall x : S \bullet x \leqslant \max S)$.

7. Orders.

$\mathrm{partial\_order}\ X$	The set of partial orders on X: $\widehat{=} \{R : X \rel X \mid \forall x, y, z : X \bullet x\ R\ x \land x\ R\ y \land y\ R\ x \implies x = y \land x\ R\ y \land y\ R\ z \implies x\ R\ z\}$.

$\mathrm{total\_order}\ X$	The set of total orders on X: $\hat{=} \{R : \mathrm{partial\_order} \mid \forall x, y : X \bullet x\,R\,y \vee y\,R\,x\}$.
$\mathrm{monotonic}\ X <_X$	The set of functions from X to X that are monotonic with respect to the order $<_X$ on X: $\hat{=} \{f : X \mathrel{\text{⇸}} X \mid \forall x, y : X \bullet x <_X y \Rightarrow f(x) <_X f(y)\}$.

8. Sequences.

Let a, b be elements of sequences, A, B be sequences and m, n be natural numbers.

$\mathrm{seq}\ X$	The set of sequences whose elements are drawn from X: $\hat{=} \{A : \mathbb{N} \mathrel{\text{⇻}} X \mid \mathrm{dom}\ A = 1 \mathrel{..} \#A\}$.
$\langle\rangle$	The empty sequence $\emptyset$.
$\mathrm{seq}_1\ X$	The set of non-empty sequences: $\hat{=} \mathrm{seq}\ X \setminus \{\langle\rangle\}$
$\langle a_1, \ldots, a_n\rangle$	$\hat{=} \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$.
$\langle a_1, \ldots, a_n\rangle \frown \langle b_1, \ldots, b_m\rangle$	Concatenation: $\hat{=} \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$, $\langle\rangle \frown A = A \frown \langle\rangle = A$.
$\mathrm{head}\ A$	The first element of a non-empty sequence: $A \neq \langle\rangle \Rightarrow \mathrm{head}\ A = A(1)$.
$\mathrm{last}\ A$	The final element of a non-empty sequence: $A \neq \langle\rangle \Rightarrow \mathrm{last}\ A = A(\#A)$.
$\mathrm{tail}\ A$	All but the head of a sequence: $\mathrm{tail}(\langle x\rangle \frown A) = A$.
$\mathrm{front}\ A$	All but the last of a sequence: $\mathrm{front}(A \frown \langle x\rangle) = A$.
$\mathrm{rev}\ \langle a_1, a_2, \ldots, a_n\rangle$	Reverse: $\hat{=} \langle a_n, \ldots, a_2, a_1\rangle$, $\mathrm{rev}\ \langle\rangle = \langle\rangle$.
$\frown/AA$	Distributed concatenation: given $AA : \mathrm{seq}(\mathrm{seq}(X))$, $\hat{=} AA(1) \frown \ldots \frown AA(\#AA)$, $\frown/\langle\rangle = \langle\rangle$.
$\mathbin{;}/AR$	Distributed relational composition: given $AR : \mathrm{seq}\ (X \leftrightarrow X)$, $\hat{=} AR(1) \mathbin{;} \ldots \mathbin{;} AR(\#AR)$, $\mathbin{;}/\langle\rangle = \mathrm{id}\ X$.
$\oplus/AR$	Distributed overriding: given $A : \mathrm{seq}\ (X \mathrel{\text{⇸}} Y)$, $\hat{=} AR(1) \oplus \ldots \oplus AR(\#AR)$, $\oplus/\langle\rangle = \emptyset$.
$\mathrm{squash}\ f$	Convert a finite function, $f : \mathbb{N} \mathrel{\text{⇻}} X$, into a sequence by squashing its domain. That is, $\mathrm{squash}\ \emptyset = \langle\rangle$, and if $f \neq \emptyset$ then $\mathrm{squash}\ f = \langle f(i)\rangle \frown \mathrm{squash}(\{i\} \mathbin{\text{⩤}} f)$ where $i = \min(\mathrm{dom}\ f)$.
$S \upharpoonleft A$	Index restriction: $\hat{=} \mathrm{squash}(S \lhd A)$.
$A \upharpoonright T$	Sequence restriction: $\hat{=} \mathrm{squash}(A \rhd T)$.
$\mathrm{disjoint}\ AS$	Pairwise disjoint: given $AS : \mathrm{seq}\ (\mathbb{P}\ X)$, $\hat{=} (\forall i, j : \mathrm{dom}\ AS \bullet i \neq j \Rightarrow AS(i) \cap AS(j) = \emptyset)$.
$AS\ \underline{\mathrm{partitions}}\ S$	$\hat{=} \mathrm{disjoint}\ AS \wedge \bigcup \mathrm{ran}\ AS = S$.
$A\ \underline{\mathrm{in}}\ B$	Contiguous subsequence: $\hat{=} (\exists C, D : \mathrm{seq}\ X \bullet C \frown A \frown D = B)$.

9. Bags.

$\mathrm{bag}\ X$	The set of bags whose elements are drawn from X: $\hat{=} X \mathrel{\text{⇸}} \mathbb{N}_1$
$\mathrm{items}\ s$	The bag of items contained in the sequence s: $\hat{=} \{x : \mathrm{ran}\ s \bullet x \mapsto \#\{i : \mathrm{dom}\ s \mid s(i) = x\}\}$

Schema Notation

Schema definition: a schema groups together some declarations of variables and a predicate relating these variables. There are two ways of writing schemas: vertically, for example

```
S ──────────────────
│ x : ℕ
│ y : seq ℕ
├─────────────
│ x ≤ #y
└───────────────────
```

or horizontally, for the same example

$S \mathrel{\widehat{=}} [\ x\colon \mathbb{N};\ y\colon \mathrm{seq}\ \mathbb{N} \mid x \le \#y\ ].$

Use in signatures after $\forall$, λ, $\{\ldots\}$, etc.:

$$(\forall S \bullet y \ne \langle\rangle) \mathrel{\widehat{=}} (\forall x\colon \mathbb{N};\ y\colon \mathrm{seq}\ \mathbb{N} \mid x \le \#y \bullet y \ne \langle\rangle).$$

Schemas as types: when a schema name S is used as a type it stands for the set of all objects described by the schema, $\{S\}$. For example, $w : S$ declares a variable w with components x (of type $\mathbb{N}$) and y (of type $\mathrm{seq}\ \mathbb{N}$) such that $x \le \#y$.

Projection functions: the component names of a schema may be used as projection (or selector) functions. For example, given $w : S$, $w.x$ is w's x component and $w.y$ is its y component; of course, the following predicate holds: $w.x \le \#w.y$. Additionally, given $w : X \nrightarrow S$, $w ⨾ (\lambda S.x)$ is a function $X \nrightarrow \mathbb{N}$, etc.

θS The tuple formed from a schema's variables: for example, θS is (x, y). Where there is no risk of ambiguity, the θ is sometimes omitted, so that just "S" is written for "(x, y)".

pred S The predicate part of a schema: e.g. $\mathrm{pred}\ S$ is $x \le \#y$.

Inclusion A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must be of the same type) and the predicates of S and T are conjoined. For example,

```
T ──────────────────
│ S
│ z : ℕ
├─────────────
│ z < x
└───────────────────
```

is

```
────────────────────
│ x, z : ℕ
│ y : seq ℕ
├─────────────
│ x ≤ #y ∧ z < x
└───────────────────
```

$S \mid P$ The schema S with P conjoined to its predicate part. E.g., $(S \mid x>0)$ is $[\ x\colon\mathbb{N}; y\colon \mathrm{seq}\ \mathbb{N} \mid x \le \#y \land x>0\ ]$.

$S\ ;\ D$ The schema S with the declarations D merged with the declarations of S. For example, $(S\ ;\ z\colon\mathbb{N})$ is $[\ x, z\colon\mathbb{N};\ y\colon \mathrm{seq}\ \mathbb{N} \mid x \le \#y\ ]$.

$S[new/old]$ Renaming of components: the schema S in which the component old has been renamed to new both in the declaration and at its every free occurrence in the predicate. For example, $S[z/x]$ is $[\ z\colon\mathbb{N};\ y\colon \mathrm{seq}\ \mathbb{N} \mid z \le \#y\ ]$ and $S[y/x, x/y]$ is $[\ y\colon\mathbb{N};\ x\colon \mathrm{seq}\ \mathbb{N} \mid y \le \#x\ ]$.

In the second case above, the renaming is simultaneous.

Decoration Decoration with prime, subscript, superscript, etc.; systematic renaming of the components declared in the schema. For example, S' is $[\,x':\mathbb{N};\ y':\mathrm{seq}\ \mathbb{N} \mid x' \leqslant \#y'\,]$.

$\neg S$ The schema S with its predicate part negated. E.g., $\neg S$ is $[\,x:\mathbb{N};\ y:\mathrm{seq}\ \mathbb{N} \mid \neg(x \leqslant \#y)\,]$.

$S \wedge T$ The schema formed from schemas S and T by merging their declarations (see inclusion above) and conjoining (and-ing) their predicates. Given $T \mathrel{\widehat{=}} [\,x:\mathbb{N};\ z:\mathbb{P}\ \mathbb{N} \mid x \in z\,]$, $S \wedge T$ is

$$\begin{array}{|l}
x : \mathbb{N} \\
y : \mathrm{seq}\ \mathbb{N} \\
z : \mathbb{P}\ \mathbb{N} \\
\hline
x \leqslant \#y \wedge x \in z
\end{array}$$

$S \vee T$ The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. For example, $S \vee T$ is

$$\begin{array}{|l}
x : \mathbb{N} \\
y : \mathrm{seq}\ \mathbb{N} \\
z : \mathbb{P}\ \mathbb{N} \\
\hline
x \leqslant \#y \vee x \in z
\end{array}$$

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking pred $S \Rightarrow$ pred T as the predicate. E.g., $S \Rightarrow T$ is

$$\begin{array}{|l}
x : \mathbb{N} \\
y : \mathrm{seq}\ \mathbb{N} \\
z : \mathbb{P}\ \mathbb{N} \\
\hline
x \leqslant \#y \Rightarrow x \in z
\end{array}$$

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking pred $S \Leftrightarrow$ pred T as the predicate. E.g., $S \Leftrightarrow T$ is

$$\begin{array}{|l}
x : \mathbb{N} \\
y : \mathrm{seq}\ \mathbb{N} \\
z : \mathbb{P}\ \mathbb{N} \\
\hline
x \leqslant \#y \Leftrightarrow x \in z
\end{array}$$

$S \setminus (v_1,\ v_2,\ \ldots,\ v_n)$
Hiding: the schema S with the variables v_1, $v_2, \ldots$, and v_n hidden: the variables listed are removed from the declarations and are existentially quantified in the predicate. E.g., $S \setminus x$ is $[\,y:\mathrm{seq}\ \mathbb{N} \mid (\exists x:\mathbb{N} \bullet x \leqslant \#y)\,]$. (We omit the parentheses when only one variable is hidden.) A schema may be specified instead of a list of variables; in this case the variables declared in that schema are hidden. For example, $(S \wedge T) \setminus S$ is

$$\begin{array}{|l}
z : \mathbb{P}\ \mathbb{N} \\
\hline
(\exists\ x:\ \mathbb{N};\ y:\ \mathrm{seq}\ \mathbb{N} \bullet \\
\quad x \leqslant \#y \wedge x \in z)
\end{array}$$

$S \upharpoonright (v_1, v_2, \ldots, v_n)$
Projection: The schema S with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden: the variables removed from the declarations are existentially quantified in the predicate. E.g., $(S \wedge T) \upharpoonright (x, y)$ is

```
┌──────────────────
│ x : ℕ
│ y : seq ℕ
├──────────────
│ (∃ z : ℙ ℕ •
│   x ≤ #y ∧ x ∈ z)
└──────────────────
```

As for hiding above, we may project a single variable with no parentheses or the variables in a schema.

The following conventions are used for variable names in those schemas which represent operations on some state:

undashed	state before,
dashed ("′")	state after,
ending in "?"	inputs to (arguments for),
ending in "!"	outputs from (results of) the operation.

The following schema operations only apply to schemas following the above conventions.

pre S
Precondition: all the state after components (dashed) and the outputs (ending in "!") are hidden. E.g. given

```
S──────────────────
│ x?, s, s′, y! : ℕ
├──────────────
│ s′ = s-x? ∧ y! = s
└──────────────────
```

pre S is

```
┌──────────────────
│ x?, s : ℕ
├──────────────
│ (∃ s′, y! : ℕ •
│  s′ = s-x? ∧ y! = s)
└──────────────────
```

post S
Postcondition: this is similar to precondition except all the state before components (undashed) and inputs (ending in "?") are hidden. (Note that this definition differs from some others, in which the "postcondition" is the predicate relating all of initial state, inputs, outputs, and final state.)

$S \oplus T$
Overriding:
$\mathrel{\widehat{=}} (S \wedge \neg \mathrm{pre}\ T) \vee T.$
For example, given S above and

```
T──────────────────
│ x?, s, s′ : ℕ
├──────────────
│ s < x? ∧ s′ = s
└──────────────────
```

S ⊕ T is

```
┌──────────────────
│ x?, s, s′, y! : ℕ
├──────────────
│ (s′ = s-x? ∧ y! = s ∧
│  ¬(∃ s′: ℕ •
│     s < x? ∧ s′ = s))
│ ∨ (s < x? ∧ s′ = s)
└──────────────────
```

which simplifies to

```
┌──────────────────
│ x?, s, s′, y! : ℕ
├──────────────
│ (s′ = s-x? ∧ y! = s ∧
│  s ≥ x?) ∨
│ (s < x? ∧ s′ = s)
└──────────────────
```

$S \fatsemi T$ Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the state-after components of S and the state-before components of T that have a basename* in common, rename both to new variables, take the schema which is the "and" (∧) of the resulting schemas, and hide the new variables. E.g., $S \fatsemi T$ is

$$\begin{array}{|l} x?, s, s', y! : \mathbb{N} \\ \hline (\exists s_0 : \mathbb{N} \; . \\ \quad s_0 = s - x \land y! = s \land \\ \quad s_0 < x? \land s' = s_0) \end{array}$$

* basename is the name with any decoration ("'", "!", "?", etc.) removed.

$S \gg T$ Piping: this schema operation is similar to schema composition; the difference is that, rather than identifying the state after components of S with the state before components of T, the output components of S (ending in "!") are identified with the input components of T (ending in "?") with the same basename.

The following conventions are used for prefixing of schema names:

ΔS	change of <u>before</u> to <u>after</u> state,
ΞS	no change of state,
ΦS	framing schema for definition of further operations.

For example

$$\begin{array}{lcl} \Delta S & \mathrel{\widehat=} & S \land S' \\ \Xi S & \mathrel{\widehat=} & \Delta S \mid \theta S = \theta S' \\ \Phi S & \mathrel{\widehat=} & \Delta S \mid y = y' \\ S_{OP} & \mathrel{\widehat=} & \Phi S \mid x' = 0 \end{array}$$

Other Definitions

Axiomatic definition: introduces global declarations which satisfy one or more predicates for use in the entire document.

$$\begin{array}{|l} \text{declaration(s)} \\ \hline \text{predicate(s)} \end{array}$$

or horizontally: $D \mid P$

Generic constant: introduces generic declarations parameterised by sets A, B, etc. which satisfy the given predicates.

$$\begin{array}{|l|} \hline [A, B, ...] \\ \text{declaration(s)} \\ \hline \text{predicate(s)} \\ \hline \end{array}$$

Generic schema definition: introduces generic schema parameterised by sets A, B, etc. When used subsequently, the schema should be instantiated (e.g. S[X, Y, ...]).

$$\begin{array}{|l|} S[A, B, ...] \\ \text{declaration(s)} \\ \hline \text{predicate(s)} \\ \hline \end{array}$$