


THE FORMAL DOCUMENTATION OF
A BLOCK STORAGE SERVICE

by

Roger Gimson

ACQUISITION NO. /	DATE 25 FEB 2002
UNIVERSITY OF OXFORD	
 303397029-	

Technical Monograph PRG-62

August 1987

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

Copyright © 1987 Roger Gimson

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

The Formal Documentation of a Block Storage Service

Roger Gimson

Abstract

The formal documentation for a low-level data storage service is presented. The service allows blocks of data to be stored on behalf of clients in a distributed system. The documentation includes both a User Manual, presenting the clients' view of the service, and an Implementor Manual, describing how the service may be implemented. It is called formal documentation because, as well as informal text giving the conventional overviews to the casual reader, it includes precise specifications of the behaviour of the service, written in the formal specification language Z.

Though applied here to the example of a block storage service, the illustrated style of documentation can equally well be applied to describing any such system components. This style has been developed as part of a project on designing and specifying components of a distributed operating system. The monograph includes a discussion of the design choices considered for the service, and the experience gained through its design, implementation and documentation.

Contents

	Introduction	5
Chapter 1.	Block Storage Service - User Manual	7
Chapter 2.	Block Storage Service - Implementor Manual	35
Chapter 3.	Discussion and Experience	87
	Acknowledgements	99
	References	100
Appendix A.	Index of formal definitions	101
Appendix B.	Glossary of Z notation	104

Introduction

A distributed operating system consists of a number of separate services connected to each other and to clients by a network. This monograph describes one such service, the Block Storage Service. It allows clients to store and retrieve fixed size blocks of data. The service only provides low-level data storage facilities, but can form the basis for one or more independent higher-level file storage services.

The service is presented as a pair of manuals. The first, the "User Manual", describes the service as seen from the outside, by a client of the service. An abstract view of the state of the service is given. Operations which may be performed on the service by a client are described in terms of changes in this abstract state.

The second, the "Implementor Manual", describes how the service works internally. A concrete view of the state of the service is given in terms of the components from which the implementation is composed. This concrete state has a well-defined relationship to the abstract state described in the first manual. Each of the operations that can be performed on the service is redefined in terms of changes to the components of the concrete state.

Apart from presenting the design of a particular service, the monograph is intended to illustrate how rigor can be introduced into the documentation of software systems. It can also be seen as an example of how it is possible to present formal specifications of system components in a style more familiar to the programmer. Only by achieving an appropriate balance between formality and accessibility of presentation can we hope that these techniques will be more widely accepted by computing practitioners.

The final chapter is a discussion based on the experience gained by formally designing, specifying and implementing the service.

The Block Storage Service was developed as a part of the Distributed Computing Software Project which began at the Oxford University Programming Research Group in 1982. The goal of the project has been to construct and publish the design of services in a loosely-coupled distributed operating system, based on the model of autonomous clients having access to a number of shared devices.

A fundamental objective of the project has been the use of mathematical techniques for program specification to assist the design, development and presentation of distributed system services. The formal notation used throughout has been the specification language Z, which has been undergoing development at the Programming

Research Group over the same period. The project has therefore been a continuing practical test of the application of Z to system specification.

The first phase of the project resulted in the specification of a number of services, including an earlier version of the User Manual for the Block Storage Service [1]. The presentation has been subsequently improved into the form shown here, and an Implementor Manual for the service has been developed. The formal notation Z, used throughout this document, is defined in [2-7]. A glossary of the notation is included here as an appendix. Common parts of services (e.g. accounting and access control) have been combined together into a separate document known as the "Common Service Framework" (included in [8]). The descriptions presented here make use of some definitions from that document.

Chapter 1

Block Storage Service - User Manual

1	Introduction
2	Service state
2.1	Blocks
2.2	Overall state
3	Operation parameters
3.1	Block-specific operations
3.2	Key-linked operations
4	Reports
5	Operation definitions
5.1	Client operations
	Create
	Read
	Status
	Destroy
	Replace
	SetExpiry
	GetIds
	GetCount
5.2	Manager operations
	Scavenge
	Profile
6	Service charges
7	Complete service

1 Introduction

The Block Storage Service provides low-level data storage facilities. Clients (typically other services or application programs) may create, access and destroy fixed-size blocks of data by invocation of the service operations. A block identifier, chosen by the service, is used to identify a particular block. A unique identifier is given to each block, and once a block has been created its identity cannot be changed.

Blocks have a fixed expiry time, chosen by the client, and will be destroyed without warning on reaching the given time.

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

As well as operations on individual blocks, the service also provides some operations to help clients keep track of their block usage, and further operations for the management of the storage provided by the service.

General features shared with other services are described in the "Common Service Framework" (contained in [8]). They will only be summarised where appropriate in this manual. In particular, the following types are common to all services, and will not be defined further here:

[*Byte*, *UserNum*, *Time*, *Money*, *Report*, *Op*]

Implementation-specific constants are shown in italics (e.g. *BlockSize*). Their actual values should be made available to users of a particular implementation, but are not included here.

There is also a given finite set of block *identifiers*.

[BlockId]

An identifier (*id*) will be issued by the service when a block is created. This becomes the client's reference to the block and any subsequent operations on the block will require this identifier.

2.2 Overall state

The service state records all currently stored blocks according to their identifiers. It also contains a finite set of new block ids which have not yet been issued. The schema *SS* denotes the state of the storage service at any particular moment.

SS
$blocks : BlockId \rightarrow Block$ $newids : \mathbb{F} BlockId$
<hr/> $\#blocks \leq MaxBlocks$ $newids \cap \text{dom } blocks = \emptyset$ $NullId \notin (\text{dom } blocks \cup newids)$

There is an implementation-specific limit (*MaxBlocks*) imposed on the total number of blocks that can be stored at any one time. The set of new ids never contains identifiers of existing blocks. The service guarantees never to issue a special identity (*NullId*); this id can therefore be used by clients' applications to indicate "no block".

Initially, when the service is started for the first time, there are no stored blocks, and all block ids except *NullId* are potentially available for issue. This is modelled as an operation with only a resultant (dashed) state.

$InitSS$
SS'
<hr/> $blocks' = \emptyset$ $newids' = BlockId \setminus \{NullId\}$

3 Operation parameters

Each service operation can only be performed by an authentic client. Authentication of clients is assumed to be performed outside the service (see "Common Service Framework" in [8]). After authentication, the client number is available as an implicit parameter of every operation, and so is the time at which the operation occurs.

Explicit input and output parameters are denoted by names ending in ? for input and ! for output. Every operation provides at least an explicit output report (report!) indicating its outcome.

ϕBasicParams	
clientnum :	UserNum
now :	Time
report! :	Report

Changes in the state of the service caused by operations all conform to the following general schema in which the state of the service before an operation (SS) is related to that after the operation (SS') and to the basic parameters of the operation.

ΔSS	
SS	
SS'	
ϕBasicParams	
newids' = newids \ dom blocks'	

It is a constraint on 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 storage service is left unchanged by an operation.

$$\cong SS \quad \cong \quad \Delta SS \mid \emptyset SS' = \emptyset SS$$

Some operations take parameters which denote a count of blocks (some number in the range from zero up to at least the maximum number of blocks), a set of block identifiers or a sequence of block information of limited length. The following sets are defined here for convenience.

3.2 Key-linked operations

Some operations are designed to operate over a potentially large set of values (such as all current block identifiers). Such operations are designed to allow the set in question to be traversed in several operation calls. This may be necessary to limit either the size of output parameters or the execution time of any particular call.

In this service, some operations require the traversal of a potentially large set of ids.

$$xs : F \text{ BlockId}$$

The operation itself is designed to traverse only a subset of xs , and repeated calls of it may be necessary to construct xs as the union of the individually traversed parts. The execution of the separate operations is related by passing a *key* parameter from one call to the next, taken from the given set of all keys:

[Key]

Each such operation has an input *key?* parameter (*key?*) and an output *key!* parameter (*key!*) and affects a subset of xs ($subxs$). To construct xs , the client first calls the operation with a special key *StartKey*:

$$\text{Operation} \mid \text{key?} = \text{StartKey}$$

The client then continues to call the operation repeatedly, supplying as the new *key* in each case the *key* returned by the previous call. For example, the i^{th} call would be:

$$\begin{aligned} \text{Operation} \mid \text{key?} &= \text{key}_i \quad \wedge \\ \text{key!} &= \text{key}_{i+1} \quad \wedge \\ \text{subxs} &= \text{subxs}_i \end{aligned}$$

Finally, the special key *EndKey* will be returned to indicate that no more calls need be made.

$$\text{Operation} \mid \text{key!} = \text{EndKey}$$

At that point, providing the set xs has remained constant, and not been affected by other operations on the service:

$$xs = \bigcup_i \text{subxs}_i$$

A key is itself to be regarded as standing for a set of block identifiers, using some implementation-specific representation (denoted by the function `KeySet`). The special keys (`StartKey` and `EndKey`) denote the set of no block ids and the set of all block ids respectively.

$\text{KeySet} : \text{Key} \rightarrow \mathbf{F} \text{BlockId}$ <hr/> $\text{KeySet}(\text{StartKey}) = \emptyset$ $\text{KeySet}(\text{EndKey}) = \text{BlockId}$

Each key value, passed from one call to the next, stands for all the ids that have been traversed so far (including possibly many that are not in $\times s$).

The following framing schema is used to simplify the definition of such key-linked operations.

$\text{Key} \xrightarrow{\quad} \text{Key}$
$\begin{aligned} \text{key?} & : \text{Key} \\ \text{key!} & : \text{Key} \\ \times s & : \mathbf{F} \text{BlockId} \\ \text{sub}\times s & : \mathbf{F} \text{BlockId} \end{aligned}$ <hr/> $\begin{aligned} \text{KeySet}(\text{key?}) & \subset \text{KeySet}(\text{key!}) \\ \text{sub}\times s & = (\text{KeySet}(\text{key!}) \setminus \text{KeySet}(\text{key?})) \cap \times s \end{aligned}$

On each key-linked call, the set of ids denoted by the output `key` is strictly larger than the set denoted by the input `key`. Since the set of all ids is finite, this implies that eventually the `EndKey` must be reached, and all potential ids traversed. The difference between the sets associated with the two keys indicates the subset of $\times s$ involved in the particular call.

4 Reports

Each service operation is specified by giving a definition of its successful execution, then augmenting this with the potential reasons for lack of success. 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.

```

Success _____
|
|   report! : Report
|
|_____
|
|   report! = 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 will the operation succeed.

In each of the following cases, the state of the service remains unchanged if an error occurs.

NoSuchBlock is given if there is no block stored with identity id?.

```

NoSuchBlock _____
|
|   ≡SS
|   id? : BlockId
|
|_____
|
|   id? ∉ dom blocks
|   report! = NoSuchBlockReport
|_____
  
```

NoSpace indicates that a new block cannot be created when the storage capacity of the service is full.

NoSpace $\equiv SS$
$\#blocks = MaxBlocks$ $report! = NoSpaceReport$

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

NotOwner $\equiv SS$ $\emptyset Block$
$block.owner \neq clientnum$ $report! = NotOwnerReport$

BadKey indicates an input key has been provided which does not denote a valid id set.

BadKey $\equiv SS$ $key? : Key$
$key? \notin \text{dom } KeySet \setminus \{EndKey\}$ $report! = BadKeyReport$

NotManager is given on an attempt by some other client to perform an operation that may only be performed by the service manager.

NotManager $\equiv SS$
$clientnum \neq Manager$ $report! = NotManagerReport$

5 Service operations

On the following pages appear descriptions of the Block Storage Service operations. They are grouped into two sections: those that can be performed by ordinary clients, and those that can be performed only by the service manager. Each operation description has three parts.

The **Abstract** gives a procedure heading for the operation, with parameter definitions, as it might appear in a programming language. The correspondence between this procedure, and an implementation of it in a real programming language, should be obvious and direct. Each parameter is given a name ending with either ? for an input or ! for an output. A short informal description of the operation accompanies the procedure heading.

The **Definition** section mathematically defines the successful execution of the operation. It does this by giving a schema which includes as a component every formal parameter of the procedure heading, either explicitly or as components of included schemas (such as `report!` in `ASS`). A short explanation accompanies the schema.

The **Reports** section provides a definition of the total operation, including the possible error reports that may be obtained from its invocation. The errors are specified by a set of error schemas, as already defined, whose names are chosen to reflect the reports they return. Schema overriding (\oplus) is used to define an ordering of potential error outcomes. This means that the later errors in a sequence of overrides will be produced, if appropriate, rather than earlier ones. The successful outcome, which appears first in the definition, will only occur if none of the error conditions are satisfied.

5.1 Client operations

There are eight operations the ordinary client may ask the service to perform:

- Create — create a new block and store it
- Read — read the data of a block
- Status — obtain the status of a block
- Destroy — remove a block from the service
- Replace — replace one block with another
- SetExpiry — change the expiry time of a block
- GetIds — obtain the identities of blocks currently owned by the client
- GetCount — obtain the number of blocks currently owned by the client.

CREATE

Abstract

```

Create (expiry? : Time;
       data?   : BlockData;
       id!    : BlockId;
       report! : Report)

```

A block is created to store the given `data?` until the given `expiry?` time, and is stored by the service under the new identity `id!`.

Definition

<pre> Create_{success} ΔSS φNewBlock expiry? : Time data? : BlockData id! : BlockId newblock.expires = max {now, expiry?} newblock.data = data? blocks' = blocks ∪ {id! ↦ newblock} </pre>
--

The expiry time of the block is set to be the requested time, or the current time, whichever is later. This ensures that a block cannot expire 'before' it has been created. Its owner and creation times are defined (by `φNewBlock`) to be the invoking client and the current time respectively. The new block is stored with a unique identity.

Reports

$$\text{Create} \hat{=} (\text{Create}_{\text{success}} \wedge \text{Success}) \bullet \text{NoSpace}$$

For success, there must be enough storage space left in the service.

READ

Abstract

```

Read (id?      : BlockId;
      data!    : BlockData;
      report!  : Report)
    
```

The data is returned for the block stored with identity id?.

Definition

$Read_{success} \equiv SS$ $\phi Block$ id? : BlockId data! : BlockData <hr style="width: 50%; margin-left: 0;"/> data! = block.data
--

The service is unchanged by this operation. The data of the stored block is returned.

Reports

$$Read \hat{=} (Read_{success} \wedge Success) \oplus NoSuchBlock$$

For success, the block must already be stored by the service.

STATUS

Abstract

```

Status (id?      : BlockId;
       owner!   : UserNum;
       created! : Time;
       expires! : Time;
       report!  : Report)

```

Status information (owner, and times of creation and expiry) is returned for the block stored with identity id?.

Definition

<pre> Status_{success} ≡SS ⊕Block id? : BlockId owner! : UserNum created! : Time expires! : Time owner! = block.owner created! = block.created expires! = block.expires </pre>
--

The service is unchanged by this operation. The owner, creation time and expiry time attributes of the stored block are returned.

Reports

```

Status ≙ (StatusSUCCESS ^ Success)
        ● NoSuchBlock

```

For success, the block must already be stored by the service.

DESTROY**Abstract**

```

Destroy (id?      : BlockId;
        report!  : Report)

```

The block stored with identity $id?$ is removed from the service.

Definition

$\text{Destroy}_{\text{success}}$
ΔSS
ϕBlock
$id? : \text{BlockId}$
$\text{blocks}' = \{id?\} \triangleleft \text{blocks}$

The block is removed from the set of stored blocks.

Reports

$$\text{Destroy} \hat{=} (\text{Destroy}_{\text{success}} \wedge \text{Success})$$

- $\bullet \text{NotOwner}$
- $\bullet \text{NoSuchBlock}$

For success, the block must already be stored by the service and the client must be the owner of the block.

REPLACE

Abstract

```

Replace (id?      : BlockId;
        data?    : BlockData;
        id!      : BlockId;
        report!  : Report)

```

The block stored with identity *id?* is replaced by one with the given *data?*. The identity of the new block is returned.

Definition

<pre> Replace_{success} ΔSS ϕBlock ϕNewBlock id? : BlockId data? : BlockData id! : BlockId newblock.expires = block.expires newblock.data = data? blocks' = ({id?} ◀ blocks) ∪ {id! ↦ newblock} </pre>
--

The new block has the same expiry as the old one, but contains the new data. Its owner and creation times are defined (by $\phi\text{NewBlock}$) to be the invoking client and the current time respectively. The old block is removed and the new one stored under its new identity.

Reports

```

Replace ≐ (Replacesuccess ^ Success)
          ● NotOwner
          ● NoSuchBlock

```

For success, the block must already be stored by the service and the client must be the owner of the block.

SETEXPIRY**Abstract**

```

SetExpiry (id?      : BlockId;
           expiry?  : Time;
           report!  : Report)

```

The block stored with identity $id?$ is changed to have the new expiry time. Its identity is not changed.

Definition

<pre> SetExpiry_{success} ΔSS φBlock id? : BlockId expiry? : Time newblock : Block newblock.owner = block.owner newblock.created = block.created newblock.expires = max {now, expiry?} newblock.data = block.data blocks' = blocks ⊙ {id? ↦ newblock} </pre>
--

The block is replaced by one having the same identity and attributes, except that the expiry time is changed to the given value, or the current time, whichever is later.

Reports

```

SetExpiry ≅ (SetExpirysuccess ^ Success)
           ⊙ NotOwner
           ⊙ NoSuchBlock

```

For success, the block must already be stored by the service and the client must be the owner of the block.

GETIDS

Abstract

```

Get Ids (key?   : Key;
        count? : BlockCount;
        key!    : Key;
        idset!  : BlockIdSet
        report! : Report)

```

Returns a set of block ids owned by the client, limited to at most `count?` entries. By key-linking (see section 3.2), all ids belonging to the client can be obtained.

Definition

$\text{Get Ids}_{\text{success}} \cong \mathbb{S}\mathbb{S}$ ΦKey $\text{key?} : \text{Key}$ $\text{count?} : \text{BlockCount}$ $\text{key!} : \text{Key}$ $\text{idset!} : \text{BlockIdSet}$ $\text{xs} : \mathbb{F} \text{BlockId}$ $\text{subxs} : \mathbb{F} \text{BlockId}$ <hr style="width: 50%; margin-left: 0;"/> $\text{xs} = \{x : \text{dom blocks} \mid \text{blocks}(x).\text{owner} = \text{clientnum}\}$ $\text{idset!} = \text{subxs}$ $\#\text{idset!} \leq \text{count?}$

The state of the service is not changed. The set of all ids to be returned (`xs`) is the set of ids of blocks owned by this client. The set of ids returned in any one call (`subxs`) is a subset of `ids` (as defined in ΦKey). The size of the returned set is limited to at most `count?` elements. (Note that this set may be empty on any particular call even if further ids remain to be returned).

Reports

$$\text{Get Ids} \cong (\text{Get Ids}_{\text{success}} \wedge \text{Success}) \oplus \text{BadKey}$$

For success, the input key must be valid.

GETCOUNT

Abstract

GetCount (count! : BlockCount;
 report! : Report)

The number of blocks currently owned by the client is returned.

Definition

$\text{GetCount}_{\text{success}}$ $\cong \text{SS}$ $\text{count!} : \text{BlockCount}$ $\text{xs} : \mathbb{F} \text{BlockId}$ <hr style="width: 50%; margin-left: 0;"/> $\text{xs} = \{ x : \text{dom blocks} \mid \text{blocks}(x).\text{owner} = \text{clientnum} \}$ $\text{count!} = \#\text{xs}$
--

The state of the service is not changed by this operation. The set of ids to be counted (xs) is the set of ids of blocks owned by this client.

Note that the count returned will include blocks which have expired but have not yet been scavenged.

Reports

$\text{GetCount} \hat{=} (\text{GetCount}_{\text{success}} \wedge \text{Success})$

There are no additional reports for this operation.

5.2 Manager operations

Operations associated with the management of the service may only be performed by a special client called the *service manager*.

There are two such operations specific to the storage service:

- Scavenge — remove expired blocks
- Profile — obtain details of block usage.

SCAVENGE

Abstract

```

Scavenge (key?   : Key;
          key!    : Key;
          count!  : BlockCount;
          report! : Report)

```

Removes a set of expired blocks from the service, returning the number removed. By key-linking (see section 3.2), all expired blocks can be scavenged.

Definition

<pre> Scavenge_{success} ΔSS ϕKey key? : Key key! : Key count! : BlockCount xs : F BlockId subxs : F BlockId xs = { x:dom blocks blocks(x).expires < now } count! = #subxs blocks' = subxs ↯ blocks </pre>
--

The set of all ids to be scavenged (xs) is the set of ids of stored blocks which have expired. The set of ids scavenged in any one call ($subxs$) is a subset of xs (as defined in $\phi Keys$). The number of blocks scavenged is returned as $count!$. Blocks scavenged in this call are removed from the service.

Reports

```

Scavenge ≐ (Scavengesuccess ^ Success)
           ⊕ BadKey
           ⊕ NotManager

```

For success, the client must be the service manager and the input key must be valid.

PROFILE

Abstract

```

Profile (key?      : Key;
        key!       : Key;
        infoseq!   : BlockInfoSeq;
        report!    : Report)

```

Returns a sequence of information about blocks stored in the service. By key-linking (see section 3.2), the information profile of all blocks can be obtained.

Definition

<pre> Profile_{success} ≡ SS ΦKey key? : Key key! : Key infoseq! : BlockInfoSeq xs : F BlockId subxs : F BlockId order : seq BlockId xs = dom blocks dom order = 1..#subxs ran order = subxs infoseq! = λ i:dom order • blocks(order(i))!BlockInfo </pre>
--

The state of the service is not changed by this operation. The set of all block ids for which information is to be returned (xs) is the set of all blocks stored in the service. The subset of these ids for which information is returned in any one call ($subxs$) is a subset of xs (as defined in ΦKey). The sequence of information returned is that of all blocks with ids in the set $subxs$ in some arbitrary ordering (given by $order$).

Reports

```

Profile ≅ (Profilesuccess ^ Success)
        ⊕ BadKey
        ⊕ NotManager

```

For success, the client must be the service manager and the input key must be valid.

6 Service charges

Clients will be held responsible for the expenses incurred by their use of the service. Expenditure will be recorded, and clients will be expected to observe any limits placed upon them (however, such limits do not form part of this service, and will be imposed separately).

The basic parameters to an operation are supplemented by two hidden parameters (since they do not appear in the procedural interface). These are an operation identifier $op?$ and the cost of executing the operation $cost!$.

$\Phi Params$	
$\Phi BasicParams$	
$op?$: Op
$cost!$: Money

There is a cost for each successful operation, which may have two components. One is the expense of performing the operation itself ($CreateCost$, $ReadCost$, etc.). The other, if present, is related to the function requested by the operation. For example, the create operation charges in advance for the storage of the given data, and the destroy operation may give a rebate (negative cost) if the block is destroyed before its expiry time.

The expense of storing a block is determined by applying a tariff function to the creation and expiry times of the block. Here is a typical block tariff function:

$BlockTariff : (Time \times Time) \rightarrow Money$
$\forall \text{ created, expires: Time} \cdot$ $BlockTariff(\text{created, expires}) =$ $BlockCost * (\text{expires} - \text{created})$

where $(_ * _) : (Money \times Time) \rightarrow Money$ is defined appropriately.

The values of $CreateCost$ etc. and the block tariff function itself may be varied; their precise values at any time will be made known separately to clients.

The cost of successfully invoking any particular operation on the storage service is defined by a tariff schema. For some operations, the cost is related to the times of

creation and expiry of an existing or a newly created block. On destroy and replace operations, a rebate is given to encourage explicit destruction or replacement, rather than letting blocks expire and be removed by the service.

SSTariff	
Φ Params	
Block	
Block'	
op? = CreateOp	\Rightarrow cost! = CreateCost + BlockTariff (created', expires')
op? = ReadOp	\Rightarrow cost! = ReadCost
op? = StatusOp	\Rightarrow cost! = StatusCost
op? = DestroyOp	\Rightarrow cost! = DestroyCost - BlockTariff (created, expires) + BlockTariff (created, now)
op? = ReplaceOp	\Rightarrow cost! = ReplaceCost - BlockTariff (created, expires) + BlockTariff (created, now) + BlockTariff (created', expires')
op? = SetExpiryOp	\Rightarrow cost! = SetExpiryCost - BlockTariff (created, expires) + BlockTariff (creeted', expires')
op? = GetIdsOp	\Rightarrow cost! = GetIdsCost
op? = GetCountOp	\Rightarrow cost! = GetCountCost
op? = ScavengeOp	\Rightarrow cost! = ScavengeCost
op? = ProfileOp	\Rightarrow cost! = ProfileCost

If an error occurs, a fixed amount may still be charged.

ErrorTariff $\hat{=}$ Φ Params | cost! = ErrorCost

These two schemas combine to form an overall tariff framing schema in which the error tariff will be charged unless the output report is successful.

Φ SSTariff $\hat{=}$ Success \Rightarrow SSTariff \wedge
 \neg Success \Rightarrow ErrorTariff

7 Complete service

The full definition of the complete Block Storage Service includes identification of clients and other components which are common to many services. It depends on a number of schemas defined in the "Common Service Framework" (in [8]).

Each separate operation in the service is given a unique operation identifier.

$$\begin{aligned} \text{SSServiceOps} \hat{=} & \\ & (\text{Create} \quad \wedge \phi\text{Params} \mid \text{op?} = \text{CreateOp} \quad) \vee \\ & (\text{Read} \quad \quad \wedge \phi\text{Params} \mid \text{op?} = \text{ReadOp} \quad) \vee \\ & (\text{Status} \quad \wedge \phi\text{Params} \mid \text{op?} = \text{StatusOp} \quad) \vee \\ & (\text{Destroy} \quad \wedge \phi\text{Params} \mid \text{op?} = \text{DestroyOp} \quad) \vee \\ & (\text{Replace} \quad \wedge \phi\text{Params} \mid \text{op?} = \text{ReplaceOp} \quad) \vee \\ & (\text{SetExpiry} \wedge \phi\text{Params} \mid \text{op?} = \text{SetExpiryOp}) \vee \\ & (\text{GetIds} \quad \quad \wedge \phi\text{Params} \mid \text{op?} = \text{GetIdsOp} \quad) \vee \\ & (\text{GetCount} \quad \wedge \phi\text{Params} \mid \text{op?} = \text{GetCountOp}) \vee \\ & (\text{Scavenge} \quad \wedge \phi\text{Params} \mid \text{op?} = \text{ScavengeOp}) \vee \\ & (\text{Profile} \quad \wedge \phi\text{Params} \mid \text{op?} = \text{ProfileOp} \quad) \end{aligned}$$

Each of these operations has a tariff associated with it.

$$\text{SSBasicOps} \hat{=} \phi\text{STariff} \wedge \text{SSServiceOps}$$

The full service state includes subsystems for a clock, accounting, statistics and controlling service access. (See the "Common Service Framework" for further details.)

$$\text{SSState} \hat{=} \text{SS} \wedge \text{Clock} \wedge \text{Accts} \wedge \text{Stats} \wedge \text{Access}$$

Service initialisation includes initialisation of the subsystems.

$$\begin{aligned} \text{InitSSState} \hat{=} & \\ & \text{InitSS} \wedge \text{InitClock} \wedge \text{InitAccts} \wedge \text{InitStats} \wedge \text{InitAccess} \end{aligned}$$

The full set of available operations includes a Null operation and those to do with the service clock, accounting, statistics and access.

SSA110ps $\hat{=}$

$$\begin{aligned} & (SSBasicOps \wedge \Delta SS \wedge \phi Clock \wedge \phi Accts \wedge \phi Stats \wedge \phi Access) \vee \\ & (Null \wedge \equiv SS \wedge \phi Clock \wedge \phi Accts \wedge \phi Stats \wedge \phi Access) \vee \\ & (ClockOps \wedge \equiv SS \wedge \Delta Clock \wedge \phi Accts \wedge \phi Stats \wedge \phi Access) \vee \\ & (AcctsOps \wedge \equiv SS \wedge \phi Clock \wedge \Delta Accts \wedge \phi Stats \wedge \phi Access) \vee \\ & (StatsOps \wedge \equiv SS \wedge \phi Clock \wedge \equiv Accts \wedge \Delta Stats \wedge \phi Access) \vee \\ & (AccessOps \wedge \equiv SS \wedge \phi Clock \wedge \equiv Accts \wedge \equiv Stats \wedge \Delta Access) \vee \\ & (NotEnabled \wedge \equiv SS \wedge \phi Clock \wedge \equiv Accts \wedge \equiv Stats \wedge \equiv Access) \end{aligned}$$

The complete specification of the service, including the possibility of a bad operation number or a non-deterministic error in the underlying implementation, is then defined as follows:

SSOps $\hat{=}$

$$\begin{aligned} & ((BadOperation \wedge \equiv SS \wedge \phi Clock \wedge \equiv Accts \wedge \equiv Stats \wedge \equiv Access) \bullet \\ & (SSA110ps \wedge \Delta SS \wedge \Delta Clock \wedge \Delta Accts \wedge \Delta Stats \wedge \Delta Access)) \\ & \vee \\ & (ServiceError \wedge \equiv SS \wedge \phi Clock \wedge \equiv Accts \wedge \equiv Stats \wedge \equiv Access) \end{aligned}$$

Operations with bad operation numbers and other service errors do not change the service state, except that the clock might tick.

Chapter 2

Block Storage Service - Implementor Manual

1	Introduction
2	Abstract state
3	Concrete state
3.1	Basic definitions
3.2	Data subsystem
3.3	Header subsystem
3.4	Bit-map subsystem
3.5	Count subsystem
3.6	Block identifiers
3.7	Consistency between subsystems
3.8	Combined concrete state
3.9	Relation to abstract state
4	Additional operations and reports
4.1	Additional operations
4.2	Key-linked operations
4.3	Error reports
5	Operation implementations
5.1	Client operations
5.2	Manager operations
6	Complete service
7	Disk layout and caching
7.1	Header blocks
7.2	Map blocks and cache
7.3	Disk layout
8	Implementation correctness
Appendix:	Iterating schemas

1 Introduction

This document is a guide to the implementation of the Block Storage Service, which provides low-level data storage facilities. It assumes that the reader is familiar with the "Block Storage Service - User Manual" which outlines the abstract specification of the service from the point of view of an external user (normally a program running on a client machine). In the following document, a concrete specification of a possible implementation of the service is presented.

In order to make the implementation more understandable, it is presented in several parts. First a number of subsystems are introduced in simplified form, each with associated suboperations, which could implement various parts of the concrete service state. These are then combined to give the overall concrete state, which is related to the abstract service state.

Some additional suboperations and some reports are introduced which are useful for defining the service operations. The implementations of the service operations themselves are then defined, largely as compositions of the suboperations relating to each of the affected subcomponents of the state. Each operation and suboperation schema may typically be implemented as a procedure in the final program.

The implementations of some of the subsystems are then further refined in order to show how their state can be stored on a disk, and to provide greater efficiency through caching and data buffering.

The specification given here is still not directly implementable. A particular programming language must be chosen by the implementor and then this design must be transcribed into the programming language (currently by hand).

The design assumes that the final programming language will be an imperative language with an inherent notion of sequences of commands or operations. There is no consideration of the use of parallelism in the implementation. The introduction of such parallelism at an appropriate level in the refinement of specifications of this kind is still an active topic of research. Our feeling is that in this particular service the parallelism could be introduced after the level of refinement presented here (i.e. after the identification of the subsystems, with associated suboperations, into which the concrete state can be decomposed).

A further simplification in the design presented here is the lack of explicit provision for handling faults in the underlying hardware, such as disk read/write errors.

2 Abstract state

As a reminder of what is to be implemented, the definition of the abstract state of the Block Storage Service, as defined in the "Block Storage Service - User Manual", is summarised here.

A block of data to be stored by the service is a fixed-size (*BlockSize*) array of bytes.

$$\text{BlockData} \hat{=} 0.. \text{BlockSize}-1 \rightarrow \text{Byte}$$

Each block also has some general information attributes.

<pre> BlockInfo owner : UserNum created : Time expires : Time ----- created ≤ expires </pre>
--

So a complete block is defined as follows in the abstract state.

<pre> Block BlockInfo data : BlockData </pre>

The overall abstract state of the service records all currently stored blocks according to their identity from a set of block identifiers (*BlockId*). Unissued ids are also recorded. There is a limit (*MaxBlocks*) on the number of blocks that can be stored by the service. The null identifier (*NullId*) is never issued.

<pre> SS blocks : BlockId → Block newids : F BlockId ----- #blocks ≤ MaxBlocks newids ∩ dom blocks = ∅ NullId ∉ (dom blocks ∪ newids) </pre>
--

Initially there are no stored blocks, and all ids except *NullId* are potentially available for issue.

InitSS _____ SS' <hr/> blocks' = \emptyset newids' = BlockId \ {NullId}
--

Some further definitions provide for block counts, sets of ids and sequences of block information.

BlockCount $\hat{=}$ 0..MaxCount

BlockIdSet $\hat{=}$ \mathbb{F} BlockId

BlockInfoSeq $\hat{=}$ { s: seq BlockInfo | #s \leq MaxInfos }

where

MaxCount \geq MaxBlocks.

3 Concrete state

The concrete state of the service is built up from several subsystems. After introducing some basic definitions which will be used throughout, the subsystems are specified as subcomponents of the system state with specific suboperations applicable to each one. The state subcomponents are then combined to give the overall concrete state.

A subsystem specification is like an abbreviated user manual. It includes a description of the state of the subsystem, its initial state, and the suboperations which may be used to change the state. Each suboperation has an abstract showing how it might appear as a procedure call in a procedural programming language.

The overall concrete state of the implementation is obtained by conjoining the states of the subsystems. In a subsequent section, the implementations of the service operations are specified as combinations of the suboperations on the individual subsystems.

3.1 Basic definitions

A byte of data is implemented as a fixed number (*ByteSize*, normally eight) of bits. By convention, these are indexed from zero upwards.

$$\begin{aligned} \text{Bit} &\cong \{0, 1\} \\ \text{Byte} &\cong 0.. \text{ByteSize}-1 \rightarrow \text{Bit} \end{aligned}$$

In this manual, we shall model data arrays as functions from fixed-size domains of index numbers (normally from zero up to a maximum value) to bytes. It is convenient to define some general functions to operate on these arrays.

$$\begin{array}{l} \text{(_ upto _)} , \\ \text{(_ from _)} , \\ \text{(_ at _)} : ((N \rightarrow \text{Byte}) \times N) \rightarrow (N \rightarrow \text{Byte}) \\ \hline \forall a : (N \rightarrow \text{Byte}); n : N \cdot \\ \quad a \text{ upto } n = (0..n-1) \triangleleft a \\ \quad a \text{ from } n = \text{succ}^n \# a \\ \quad a \text{ at } n = \text{pred}^n \# a \end{array}$$

The functions define a new array from the beginning upto (but not including) a certain position in the supplied array or from a given position to the end of the supplied array. Additionally, it is possible to move the domain of an array so that it starts at a specified offset.

Each block in the store will be identified by a block number from the set `BlockNum`. This is a finite subset of the natural numbers. There are as many different block numbers as the potential number of blocks that can be stored by the service.

$\text{BlockNum} : \mathbf{F N}$ <hr style="width: 100%;"/> $\# \text{BlockNum} = \text{MaxBlocks}$

Each block has an associated physical data-block, which holds the bulk of the data associated with the block, and a header, which holds the remainder of the block information. The number of a block will be used for identifying its associated data-block and header.

The physical disk layout may dictate that the block numbers are not contiguous. A function is defined to provide the next higher block number after a given number in the set.

$\text{next} : \text{BlockNum} \rightarrow \text{BlockNum}$ <hr style="width: 100%;"/> $\forall \text{bn} : \text{BlockNum} \mid \text{bn} \neq \max \text{BlockNum} \cdot$ $\text{next}(\text{bn}) = \min \{ n : \text{BlockNum} \mid \text{bn} < n \}$
--

3.2 Data subsystem

In the concrete state, an array of data-blocks (stored on a disk, as shown later) are used to hold the bulk of the data of the stored blocks. Each data-block can hold a fixed amount (`DataBlockSize`) of bytes relating to a particular service block.

$$\text{DataBlock} \hat{=} 0.. \text{DataBlockSize}-1 \rightarrow \text{Byte}$$

Data-blocks are indexed by the number of the block whose data they hold. The first subcomponent of the concrete state of the service is therefore the storage for the data-blocks.

```

DataBlocks _____
|   dataBlocks : BlockNum → DataBlock   |
|_____|

```

This subsystem may be in any initial state when the service is started for the first time, so the data-blocks may take any initial values.

```

InitDataBlocks _____
|   DataBlocks'   |
|_____|

```

The suboperations applicable to this subsystem are those of getting data from and putting data to the store, given the relevant block number.

GetData

```
GetData (bn? : BlockNum; datablock! : DataBlock)
```

```

GetData _____
| ΔDataBlocks   |
|   bn?         : BlockNum   |
|   datablock! : DataBlock   |
|_____|
|   datablock! = dataBlocks(bn?)   |
|_____|

```

PutData

```
PutData (bn? : BlockNum; datablock? : DataBlock)
```

```

PutData _____
| ΔDataBlocks   |
|   bn?         : BlockNum   |
|   datablock? : DataBlock   |
|_____|
|   dataBlocks' = dataBlocks * {bn? ↦ datablock?}   |
|_____|

```

Unless otherwise defined, it will be assumed that for any state S , $\Delta S \hat{=} S \wedge S'$. Note that the `GetData` operation does not define a new subsystem state. This is so that it may be conjoined in a later service operation definition either with `PutData` (which does define a new subsystem state), or with a general schema specifying no change in state.

3.3 Header subsystem

It is convenient for service users to be provided with a block size slightly larger than an exact power of two, so that they can store additional attributes with each block, such as a reference count for a file service. (A conventional size of 512 bytes might be increased to 528 bytes, for example).

Since data-blocks stored on a disk are usually a power of two in size, not all the data component of a service block will therefore fit into a single implemented data-block.

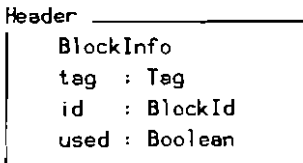
$$BlockSize > DataBlockSize$$

The remaining bytes (called the tag) are stored separately.

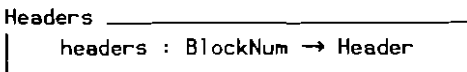
$$TagSize = BlockSize - DataBlockSize$$

$$Tag \hat{=} 0..TagSize-1 \rightarrow \text{Byte}$$

The tag is stored, along with other attributes associated with each block, in a header. The other header attributes consist of the BlockInfo and some extra information required by the implementation. This includes the block id by which a client may refer to the block and an indication of whether the block is currently being used to store data.



The next subcomponent of the concrete state of the service is then the storage for the headers, indexed by the corresponding block number.



Initially, when the service is first started, all the headers denote unused blocks.

```

InitHeaders
-----
  Headers'

  ∀ bn:BlockNum •
    headers'(bn).used = False
  
```

The suboperations applicable to this subsystem are those of getting headers from and putting them to the store.

GetHeader

GetHeader (bn? : BlockNum; header! : Header)

```

GetHeader
-----
  ΔHeaders
  bn?      : BlockNum
  header!  : Header

  header! = headers(bn?)
  
```

PutHeader

PutHeader (bn? : BlockNum; header? : Header)

```

PutHeader
-----
  ΔHeaders
  bn?      : BlockNum
  header?  : Header

  headers' = headers • {bn? ↦ header?}
  
```

As in the data subsystem, only PutHeader defines a new subsystem state.

3.4 Bit-map subsystem

It is desirable for blocks which are available for data storage to be found without reading too many headers (and hence making many disk accesses) during the search. To do this, the service implementation includes a bit-map. Each bit in the map indicates the availability of a corresponding block.

$$\text{FreeBit} \hat{=} 1 ; \text{UsedBit} \hat{=} 0$$

The next subcomponent of the concrete service state is therefore the bit-map.

```

Bitmap _____
|
| bitMap : BlockNum → Bit
|
|_____

```

Initially, when the service is first started, the bit-map shows only free (unallocated) blocks.

```

InItBitMap _____
|
| BitMap'
|_____
|
| ran bitMap' = {FreeBit}
|
|_____

```

The suboperations applicable to this subsystem are those of allocating and freeing bits, and of finding a block number corresponding to a free block.

AllocBit

AllocBit (bn? : BlockNum)

```

AllocBit _____
|
| ΔBitMap
|   bn? : BlockNum
|_____
|
| bitMap' = bitMap ⊙ {bn? ↦ UsedBit}
|
|_____

```

FreeBit

FreeBit (bn? : BlockNum)

FreeBit
Δ BitMap bn? : BlockNum
$bitMap' = bitMap \oplus \{bn? \mapsto FreeBit\}$

FindFreeBlock

FindFreeBlock (bn! : BlockNum)

FindFreeBlock
Δ BitMap bn! : BlockNum
$bitMap(bn!) = FreeBit$

This last suboperation is specified non-constructively (it doesn't say which block number is to be chosen from several candidates), and would in general involve a search through the bit-map to find a block that was marked as free. An associated error report is introduced later to cater for the case that there are no free blocks.

3.5 Count subsystem

A separate count is kept of the number of service blocks owned by each client, so that it is not necessary to scan all the block headers to extract this information.

This forms a further subcomponent of the concrete service state.

Counts
$counts : UserNum \rightarrow BlockCount$

It is assumed that the number of users is sufficiently small that a count can be held for each user (and hence a total function is used in the specification).

Initially, the counts for all clients are zero.

InitCounts <hr/> Counts' <hr/> ran counts' = {0}
--

Suboperations allow the count for a particular user to be incremented, decremented or inspected.

IncCount

IncCount (usernum? : UserNum)

IncCount <hr/> ΔCounts usernum? : UserNum <hr/> counts' = counts @ {usernum? ↦ counts(usernum?) + 1}

DecCount

DecCount (usernum? : UserNum)

DecCount <hr/> ΔCounts usernum? : UserNum <hr/> counts' = counts @ {usernum? ↦ counts(usernum?) - 1}

FetchCount

FetchCount (usernum? : UserNum; count! : BlockCount)

FetchCount <hr/> ΔCounts usernum? : UserNum count! : BlockCount <hr/> count! = counts(usernum?)

3.6 Block identifiers

The service requires the generation of unique block identifiers. For the sake of easily finding the block associated with a particular block id in the implementation, the block number is encoded within the id. The id also contains a component to provide uniqueness, since the same block number may be re-used many times for different user blocks during the lifetime of the service.

Using the clock value at the time of allocation gives uniqueness down to the granularity of the clock (assuming the clock is not allowed to run backwards!). For this implementation it is assumed that the granularity is sufficiently fine that each service operation will occur at a different time.

$$\text{BlockIdParts} \cong \text{BlockNum} \times \text{Time}$$

A special function is used to construct a block id from its components. This must be invertible, but should disguise the components so that a client is not tempted to make use of the encoded information and so become dependent on this particular implementation.

$$\left. \begin{array}{l} \text{BID} : \text{BlockIdParts} \rightarrow \text{BlockId} \\ \hline \text{ran BID} \subseteq \text{BlockId} \setminus \{\text{NullId}\} \end{array} \right\}$$

The range of BID is made a sparse subset of all possible block ids (and excludes the null id). This provides an initial barrier to attempts to use arbitrary data as block ids, and hence a limited amount of security. It is convenient to define a partial function to extract the block number from a block id.

$$\left. \begin{array}{l} \text{BIDN} : \text{BlockId} \rightarrow \text{BlockNum} \\ \hline \text{BIDN} = \text{BID}^{-1} \circ (\lambda (b, t) : \text{BlockIdParts} \cdot b) \end{array} \right\}$$

A suboperation is provided to extract the block number from the block id supplied as input to a service operation (an associated error report is introduced later to allow for a bad id).

GetBlockNum

```
GetBlockNum (id? : BlockId; bn! : BlockNum)
```

<pre>GetBlockNum id? : BlockId bn! : BlockNum</pre>
<pre>bn! = BIDN(id?)</pre>

Another suboperation constructs a new block id for a given numbered block, using the current time.

NewBlockId

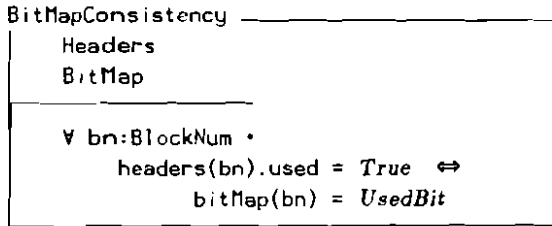
```
NewBlockId (bn? : BlockNum; id! : BlockId)
```

<pre>NewBlockId bn? : BlockNum id! : BlockId now : Time</pre>
<pre>id! = BID(bn?, now)</pre>

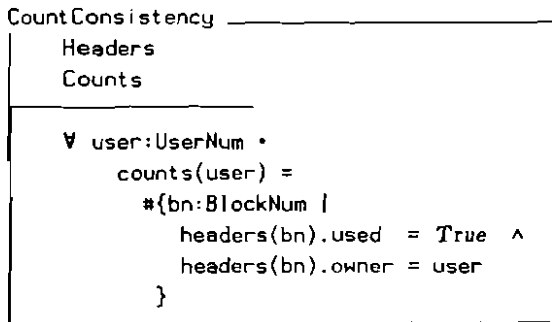
3.7 Consistency between subsystems

There are certain consistency constraints which should hold between the subcomponents in any valid concrete state. The operations as defined in this manual should preserve these constraints. However, it is possible that disk or other operational failures may compromise this consistency in an actual implementation. These constraints therefore form the basis for programs which could check the integrity of the information after a crash, and reconstruct a consistent service state.

The bit-map should reflect the usage information stored in each block header.

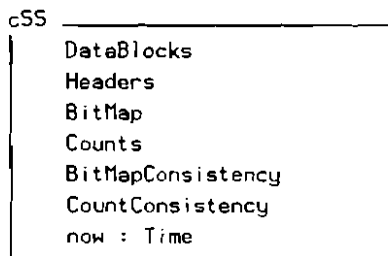


The counts should reflect the number of block headers currently in use by each user.



3.8 Combined concrete state

The complete concrete service state is obtained by combining the states of the subsystems already described, which must be consistent. The value of the service clock is also made part of the combined state.



The identifying number of the client invoking the operation (`clientnum`) and a report of the outcome of the operation (`report!`) are common parameters to all service operations.

```

φBasicParams
┌───────────────────────────┐
│ clientnum : UserNum      │
│ report!   : Report      │
└───────────────────────────┘

```

Each operation involves a potential change in the state of the service. The current time will strictly increase in value from one operation to the next (as required to generate unique identifiers).

```

ΔcSS
┌───────────────────────────┐
│ cSS                       │
│ cSS'                      │
│ φBasicParams              │
└───────────────────────────┘
│ now' > now                │
└───────────────────────────┘

```

Sometimes the complete state of the storage service (except for the current time) is left unchanged by an operation.

$$\cong cSS \cong \Delta cSS \mid \theta(cSS' \setminus now') = \theta(cSS \setminus now)$$

Some operations leave subcomponents of the state unchanged.

$$\begin{aligned} \cong \text{DataBlocks} &\cong \Delta cSS \mid \theta \text{DataBlocks}' = \theta \text{DataBlocks} \\ \cong \text{BitMap} &\cong \Delta cSS \mid \theta \text{BitMap}' = \theta \text{BitMap} \\ \cong \text{Counts} &\cong \Delta cSS \mid \theta \text{Counts}' = \theta \text{Counts} \end{aligned}$$

Initially, the service state is defined by the initial states of all the subcomponents.

```

cInitSS
┌───────────────────────────┐
│   InitDataBlocks          │
│   InitHeaders             │
│   InitBitMap              │
│   InitCounts              │
└───────────────────────────┘

```

3.9 Relation to abstract state

The state refinement step is expressed by relating the abstract user state to the concrete implementation state in the following abstraction relation.

Since the bit-maps and block counts can be derived from the block headers in the concrete state, the blocks of the abstract state can be defined entirely in terms of the data-blocks and headers from the concrete state. The identifiers available as new block ids in the abstract state depend only on the current clock value in the concrete state.

```

RelSS
┌───────────────────────────┐
│   SS                      │
│   cSS                     │
│                             │
│   blocks =                │
│   {bn      : BlockNum;     │
│     header : Header;      │
│     block  : Block |      │
│       header = headers(bn) │
│       header.used = True   │
│       block.owner  = header.owner │
│       block.created = header.created │
│       block.expires = header.expires │
│       block.data   = header.tag ∪   │
│                               (dataBlocks(bn) at TagSize) │
│       • header.id ↦ block │
│   }                        │
│   newids =                 │
│   {bn:BlockNum; t:Time | t > now • BID(bn,t)} │
└───────────────────────────┘

```

4 Additional operations and reports

In order to implement the service operations it is convenient to introduce some further suboperations. The implementation of key-linked operations is discussed, with the introduction of related suboperations. It is also necessary to define the situations in which errors will occur and the corresponding report values that will be returned.

4.1 Additional operations

The following additional operations make use of some of the suboperations previously defined to perform actions that are required by several of the subsequent service operation implementations.

A suboperation is introduced to combine the extraction of the block number from the block id and the reading of the block header (an associated error report is introduced later which checks that the given id matches that of the header and that the block is still in use). It produces the block number and the header of the block as results.

GetAttributes

```
GetAttributes (id?      : BlockId;
               bn!      : BlockNum;
               header!  : Header);
```

```
GetAttributes
┌───────────────────────────────────────────────────────────────────────────────────┐
│ GetBlockNum [id?, bn!]                                                         │
│ GetHeader  [bn!/bn?, header!]                                                 │
│ id?       : BlockId                                                           │
│ bn!       : BlockNum                                                           │
│ header!   : Header                                                             │
└───────────────────────────────────────────────────────────────────────────────────┘
```

(Note that we introduce the notation $S[x]$ to have an identical meaning to the idempotent schema renaming $S[x/x]$ in order to provide an abbreviated reminder of the name of a relevant parameter of the schema in question. Schema inclusion is used here to show that `GetAttributes` is implemented by calls on `GetBlockNum` and `GetHeader`.)

A further suboperation constructs a new block id, given the number of the block to be used. It uses this and the client's identity to initialise the header containing the attributes of the newly allocated block.

NewAttributes

```
NewAttributes (bn?      : BlockNum;
               id!      : BlockId;
               header!  : Header)
```

<pre>NewAttributes NewBlockId [bn?, id!] bn? : BlockNum id! : BlockId header! : Header clientnum : UserNum header!.owner = clientnum header!.id = id! header!.used = True</pre>
--

4.2 Key-linked operations

Keys are used to link together related calls of a particular service operation, which in conjunction potentially allow all the blocks in the service to be scanned. They are implemented via an invertible function of block numbers. Intuitively, the set of blocks denoted by a key in a key-linked operation is all those blocks with block numbers less than that obtained from applying the function to the key. The *StartKey* corresponds to the minimum block number, and the *EndKey* corresponds to no block number. The fact that not all keys correspond to block numbers gives a degree of protection against the use of arbitrary key values.

<pre>KN : Key \rightsquigarrow BlockNum KN(StartKey) = min BlockNum EndKey \notin dom KN</pre>

Keys are used to encode the starting and ending positions of scanning operations.

These operations involve iterations in which the current block number is successively incremented. The state information passed from one scan iteration to the next therefore consists of the current block number, plus an indication of whether the last block has been reached and a count of the number of blocks scanned so far.

```

Scan
-----
bn      : BlockNum
last    : Boolean
scanned : BlockCount
  
```

On each iteration, the block count is incremented and, if the last block has not been reached, the next block number is set up.

```

ΔScan
-----
Scan
Scan'

bn ≠ max BlockNum ⇒
  bn' = next(bn)
  last' = False
bn = max BlockNum ⇒
  last' = True
scanned' = scanned + 1
  
```

At the start of a scan, the scan state information is initialised using the input key (an appropriate error is defined in the next section in case the input key is invalid).

```

StartScan
-----
Scan
key? : Key

bn = KN(key?)
last = False
scanned = 0
  
```

On finishing a scan, the output key is defined according to the final block number, depending on whether the last block has been reached.

EndScan	
Scan'	
key! : Key	
<hr/>	
last' = True	⇒ key! = EndKey
last' = False	⇒ KN(key!) = bn'

4.3 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.

cSuccess	
report! : Report	
<hr/>	
report! =	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 will the operation succeed. In each of the following cases, the state of the service remains unchanged if an error occurs.

NoSuchBlock is given if there is no block stored with identity *id*?. It may result from either the input of a bad block identifier (which does not correspond to any data block) or of an identifier which denotes a block which no longer has that *id* or is no longer in use. These two cases can be separately specified as follows.

```

cBadId
  ≡ cSS
  id? : BlockId

  id? ∉ dom BIDN
  report! = NoSuchBlockReport

```

```

cMismatchedId
  ≡ cSS
  GetAttributes [id?, bn/bn!, header/header!]
  id?      : BlockId
  bn       : BlockNum
  header   : Header

  (header.id ≠ id?) ∨ (header.used = False)
  report! = NoSuchBlockReport

```

The combined error report is obtained by overriding one case with the other, since it is necessary to check for a bad id before a mismatched id can be checked.

$$\text{cNoSuchBlock} \hat{=} \text{cMismatchedId} \circledast \text{cBadId}$$

NoSpace indicates that a new block cannot be created when the storage capacity of the service is exhausted (i.e. when a search of the bit-map shows that all blocks are in use).

```

cNoSpace
  ≡ cSS

  ran bitMap = {UsedBit}
  report! = NoSpaceReport

```

NotOwner indicates an attempt to perform an operation which can destroy a block by a client who does not own the block.

```

cNotOwner
  ≡cSS
  GetAttributes [id?,bn/bn!,header/header!]
  id?      : BlockId
  bn       : BlockNum
  header   : Header

  header.owner ≠ clientnum
  report! = NotOwnerReport

```

NotManager is given if a management operation is attempted by a client who is not the service manager.

```

cNotManager
  ≡cSS

  clientnum ≠ Manager
  report! = NotManagerReport

```

BadKey indicates that a key value has been given as input which does not correspond to any block.

```

cBadKey
  ≡cSS
  key? : Key

  key? ≠ dom KN
  report! = BadKeyReport

```

In the implementation described here, no account has been taken of potential faults in the underlying hardware, such as bad disk blocks. Faults that are unrecoverable are allowed for in the abstract service specification by the catch-all error report `ServiceError`. In this event, the state of the service is specified to remain unchanged.

5 Operation implementations

The service operations are redefined here in terms of the concrete service state, making use of the suboperations from the subsystems. Each service operation is specified by conjoining the suboperations which are used in its implementation (by the use of schema inclusion).

Parameter passing between suboperations is denoted by defining some auxiliary buffer variables in the operation schema. The correspondence between the 'formal' parameters of the suboperations and the 'actual' parameters of an operation implementation is specified by renaming applied to each included suboperation schema.

As in the "User Manual", the description of each operation has three sections.

The **Abstract** section is included to reduce cross-reference with the "User Manual". It gives the procedural interface to the operation for a program running on a client machine. This will of course need to be adapted for a particular programming language.

The **Definition** section gives the formal description of the operation in terms of the concrete state together with informal details to aid the implementor. In general, the operation definitions make use of some suboperations, shown as included schemas with parameter renaming. Though there is no formal indication of the ordering amongst these suboperations, the order in which they are presented is intended to reflect the order in which they would be invoked in the implementation. The ordering is intended to ensure that each variable is defined before it is used, so leading to a procedural program.

The **Reports** section covers error conditions to produce a formal description of the total operation. As in the User Manual, schema overriding (⊗) is used to define an ordering of potential error outcomes. This means that the later errors in a sequence of overrides will be produced, if appropriate, rather than earlier ones. The successful outcome, which comes first in the definition, will only be produced if none of the error conditions are satisfied.

5.1 Client operations

There are eight operations the ordinary client may ask the service to perform:

- Create — create a new block and store it
- Read — read the data of a block
- Status — obtain the status of a block
- Destroy — remove a block from the service
- Replace — replace one block with another
- SetExpiry — change the expiry time of a block
- GetIds — obtain the identities of blocks currently owned by the client
- GetCount — obtain the number of blocks currently owned by the client.

CREATE**Abstract**

```

Create (expiry? : Time;
      data?   : BlockData;
      id!    : BlockId;
      report! : Report)

```

Definition

```

cCreatesuccess
  ΔcSS
  FindFreeBlock [bn/bn!]
  AllocBit [bn/bn?]
  NewAttributes [bn/bn?, id!, header/header!]
  PutData [bn/bn?, datablock/datablock?]
  PutHeader [bn/bn?, header/header?]
  IncCount [clientnum/usernum?]
  expiry?   : Time
  data?     : BlockData
  id!       : BlockId
  bn        : BlockNum
  datablock : DataBlock
  header    : Header
  clientnum : UserNum

  datablock = data? from TagSize
  header.tag = data? upto TagSize
  header.created = now
  header.expires = max {now, expiry?}

```

A free block is found, it is marked as allocated and a new header is created for it. The expiry time of the block is set according to the given parameter. The data is split between the header tag field and the data-block. The data and header are written to the disk, and the block count incremented for this client.

Reports

$$cCreate \hat{=} (cCreate_{success} \wedge cSuccess) \oplus cNoSpace$$

READ**Abstract**

```

Read (id?      : BlockId;
     data!    : BlockData;
     report!  : Report)

```

Definition

<pre> cRead_{success} ≡ cSS GetAttributes [id?, bn/bn!, header/header!] GetData [bn/bn?, datablock/datablock!] id? : BlockId data! : BlockData bn : BlockNum datablock : DataBlock header : Header data! = header.tag ∪ (datablock at TagSize) </pre>
--

The data is reconstructed from the header tag field and the data-block contents.

The state of the service is not changed.

Reports

$$cRead \hat{=} (cRead_{success} \wedge cSuccess) \oplus cNoSuchBlock$$

STATUS**Abstract**

```

Status (id?      : BlockId;
       owner!   : UserNum;
       created! : Time;
       expires! : Time;
       report!  : Report)

```

Definition

<pre> cStatus_{success} ≡ cSS GetAttributes [id?, bn/bn!, header/header!] id? : BlockId owner! : UserNum created! : Time expires! : Time bn : BlockNum owner! = header.owner created! = header.created expires! = header.expires </pre>

The appropriate attributes are returned as output parameters.

The state of the service is not changed.

Reports

$$\begin{aligned}
cStatus &\hat{=} (cStatus_{success} \wedge cSuccess) \\
&\quad \oplus cNoSuchBlock
\end{aligned}$$

DESTROY

Abstract

```

Destroy (id?      : BlockId;
        report!  : Report)
    
```

Definition

<pre> cDestroy_{success} ΔcSS ≡DataBlocks GetAttributes [id?,bn/bn!,header/header!] PutHeader [bn/bn?,header'/header?] FreeBit [bn/bn?] DecCount [clientnum/usernum?] id? : BlockId bn : BlockNum header : Header header' : Header clientnum : UserNum </pre> <hr style="width: 30%; margin-left: 0;"/> <pre> header'.used = False </pre>

The header and relevant bit in the bit-map are marked as being free and the block count for this client is decremented.

The data-block and new id components of the service state are not changed.

Reports

```

cDestroy ≐ (cDestroysuccess ∧ cSuccess)
           ● cNotOwner
           ● cNoSuchBlock
    
```

REPLACE**Abstract**

```

Replace (id?      : BlockId;
        data?    : BlockData;
        id!      : BlockId;
        report!  : Report)

```

Definition

```

cReplacesuccess
  ΔcSS
  ≡BitMap
  ≡Counts
  GetAttributes [id?, bn/bn!, header/header!]
  NewAttributes [bn/bn?, id!, header'/header!]
  PutData [bn/bn?, datablock/datablock?]
  PutHeader [bn/bn?, header'/header?]
  id?      : BlockId
  data?    : BlockData
  id!      : BlockId
  bn       : BlockNum
  datablock : DataBlock
  header   : Header
  header'  : Header

  header'.created = now
  header'.expires = header.expires
  header'.tag = data? upto TagSize
  datablock = data? from TagSize

```

A new header is created for the block, including a new block identity, partly from the old attributes and partly from the input parameters. The expiry time remains the same as for the old block. The new data is split between the header tag field and the data-block, which is overwritten (since the block number remains the same).

Reports

```

cReplace ≡ (cReplacesuccess ^ cSuccess)
          ⊕ cNotOwner
          ⊕ cNoSuchBlock

```

SETEXPIRY

Abstract

```

SetExpiry (id?      : BlockId;
           expiry?  : Time;
           report!  : Report)

```

Definition

<pre> cSetExpiry_{success} ΔcSS ≡DataBlocks ≡BitMap ≡Counts GetAttributes [id?, bn/bn!, header/header!] PutHeader [bn/bn?, header'/header?] id? : BlockId expiry? : Time bn : BlockNum header : Header header' : Header header'.owner ≡ header.owner header'.created ≡ header.created header'.expires ≡ max{now, expiry?} header'.tag ≡ header.tag header'.id ≡ header.id header'.used ≡ header.used </pre>

The expiry field in the block header is changed to give the block the desired expiry time. The rest of the header remains unchanged.

Reports

```

cSetExpiry ≡ (cSetExpirysuccess ^ cSuccess)
            ⊕ cNotOwner
            ⊕ cNoSuchBlock

```


GETIDS

Abstract

```

GetIds (key?   : Key;
       count? : BlockCount;
       key!    : Key;
       idset!  : BlockIdSet
       report! : Report)

```

Definition

This operation involves a scan of some of the block headers. The information passed from one scan step to the next consists of the standard scan information (defined in section 4.2), plus the accumulating id set.

Δ ScanIds Δ Scan idset : BlockIdSet idset' : BlockIdSet
--

For each relevant block header obtained from scanning the disk, the id of the block is added to the result set only if the block is in use and it is owned by the client.

GetBlockId Δ ScanIds GetHeader [bn/bn?, header/header!] bn : BlockNum header : Header clientnum : UserNum (header.used = True) \wedge (header.owner = clientnum) \Rightarrow idset' = idset \cup {header.id} (header.used \neq True) \vee (header.owner \neq clientnum) \Rightarrow idset' = idset
--

This scan operation is iterated as many times as necessary in order to accumulate all ids of blocks owned by the client, starting from the initially given block number and continuing up to either the last block header, or to a maximum number (count?) of returned ids, or after a maximum number (*MaxScan*) of blocks have been scanned.

```

EndIdsIteration
  ScanIds
  count? : BlockCount
  (last = True)      v
  (#idset = count?) v
  (scanned = MaxScan)

```

The service operation is then implemented by this iteration (see page 86 for definition of the schema while operator). Initially the set to be accumulated is empty. The final accumulated set is returned as the result.

```

cGetIdssuccess
  ≡ cSS
  ΔScanIds
  StartScan
  GetBlockId while ¬EndIdsIteration
  EndScan
  key? : Key
  count? : BlockCount
  key! : Key
  idset! : BlockIdSet
  idset = ∅
  idset! = idset'

```

Reports

$$cGetIds \hat{=} (cGetIds_{success} \wedge cSuccess) \oplus cBadKey$$

GETCOUNT**Abstract**

```
GetCount (count! : BlockCount;
         report! : Report)
```

Definition

<pre>cGetCount_{success} ≡ cSS FetchCount {clientnum/username?, count!} count! : BlockCount clientnum : UserNum</pre>
--

The number of used blocks owned by the client is returned, as determined from the stored count information.

The state of the service remains the same.

Reports

$$cGetCount \cong (cGetCount_{success} \wedge cSuccess)$$

5.2 Manager operations

Operations associated with the management of the service may only be performed by a special client called the *service manager*.

There are two manager operations specific to the storage service:

- Scavenge — remove expired blocks
- Profile — obtain details of block usage.

SCAVENGE

Abstract

```
Scavenge (key?    : Key;
          key!     : Key;
          count!  : BlockCount;
          report! : Report)
```

Definition

This operation involves a scan of some of the block headers. The information passed from one scan step to the next consists of the standard scan information (defined in section 4.2), plus a count of scavenged blocks.

```
AScanScavenge —————
|
|   ΔScan
|   scavenged : BlockCount
|   scavenged' : BlockCount
|
```

Each block which is eligible for scavenging has its header set to indicate it is free, and the corresponding bit-map bit is freed. The count of blocks used by the owner of the block is decremented. The count of scavenged blocks is incremented.

```
ScavengeBlock —————
|
|   ΔcSS
|   ≡DataBlocks
|   ΔScanScavenge
|   GetHeader [bn/bn?, header/header!]
|   PutHeader [bn/bn?, header'/header?]
|   FreeBit [bn/bn?]
|   DecCount [owner/usernum?]
|   bn      : BlockNum
|   header  : Header
|   header' : Header
|   owner   : UserNum
|
|   header'.used = False
|   owner = header.owner
|   scavenged' = scavenged + 1
|
```

Ineligible blocks are those which are not in use or have not yet expired. For these blocks the scavenge count and the state of the service remain unchanged.

```

NotExpiredBlock _____
  ≡cSS
  ΔScanScavenge
  GetHeader [bn/bn?, header/header!]
  bn      : BlockNum
  header  : Header
  now     : Time

  (header.used = False) ∨ (header.expires ≥ now)
  scavenged' = scavenged
  
```

The basic operation per block involves checking whether it has expired, and if so scavenging it. (Note that the `GetHeader` suboperation in the above two schemas need only be invoked once in the combined checking operation).

$\text{CheckBlock} \hat{=} \text{ScavengeBlock} \circ \text{NotExpiredBlock}$

This scan operation is iterated as many times as necessary in order to scavenge all blocks, starting from the initially given block number and continuing up to either the last block header, or after a maximum number (*MaxScan*) of blocks have been scanned.

```

EndScavengeIteration _____
  ScanScavenge
  _____
  (last = True) ∨ (scanned = MaxScan)
  
```

The service operation is then implemented by this iteration (see page 86 for definition of the schema `while` operator). Initially the count of scavenged blocks is zero. The final scavenge count is returned as the result.

```

cScavengesuccess
  ΔcSS
  ΔScanScavenge
  StartScan
  CheckBlock while ¬EndScavengeIteration
  EndScan
  key?   : Key
  key!   : Key
  count! : BlockCount

  scavenged = 0
  count! = scavenged'

```

Reports

$$\begin{aligned}
 \text{cScavenge} \cong & (\text{cScavenge}_{\text{success}} \wedge \text{cSuccess}) \\
 & \oplus \text{cBadKey} \\
 & \oplus \text{cNotManager}
 \end{aligned}$$

PROFILE

Abstract

```

Profile (key?      : Key;
        key!       : Key;
        infoseq!  : BlockInfoSeq;
        report!   : Report)
    
```

Definition

This operation involves a scan of some of the block headers. The information passed from one scan step to the next consists of the standard scan information (defined in section 4.2), plus the accumulating information sequence.

```

ΔScanInfo
┌───────────┴───────────┐
| ΔScan                    |
| infoseq  : BlockInfoSeq |
| infoseq' : BlockInfoSeq |
└───────────┴───────────┘
    
```

For each block whose header is scanned, the relevant header information is appended onto the sequence of block information only if the block is in use.

```

GetBlockInfo
┌───────────┴───────────┐
| ≡cSS                    |
| ΔScanInfo                |
| GetHeader [bn/bn?,header/header!] |
| bn      : BlockNum        |
| header  : Header          |
├───────────┴───────────┤
| (header.used = True) ⇒   |
|   infoseq' = infoseq ^ <header↑BlockInfo> |
| (header.used ≠ True) ⇒  |
|   infoseq' = infoseq     |
└───────────┴───────────┘
    
```

This scan operation is iterated as many times as necessary in order to accumulate all the block information, starting from the initially given block number and continuing up to either the last block header or after a maximum number (*MaxScan*) of blocks have been scanned.


```

EndInfoIteration _____
| ScanInfo _____ |
| (last = True) v (scanned = MaxScan) |
|_____|
    
```

Note that in order to ensure that the output sequence is not too long:
 $MaxScan \leq MaxInfos$

The service operation is then implemented by this iteration (see appendix for definition of the schema while operator). Initially the sequence to be accumulated is empty. The final accumulated sequence is returned as the result.

```

cProfilesuccess _____
| ≡cSS |
| ΔScanIds |
| StartScan |
| GetBlockInfo while ~EndInfoIteration |
| EndScan |
| key? : Key |
| key! : Key |
| infoseq! : BlockInfoSeq |
|_____|
| infoseq = <> |
| infoseq! = infoseq' |
|_____|
    
```

Reports

```

cProfile ≙ (cProfilesuccess ^ cSuccess)
          ● cBadKey
          ● cNotManager
    
```

6 Complete service

This section provides a combined definition of the operations of the implemented Block Storage Service. It does not include details of the implementation of service components, such as access control and accounting, which are incorporated from the "Common Service Framework".

Both in the abstract and the concrete model of the service, the basic parameters are supplemented by two *hidden* parameters, an operation identifier (op?) and the cost of executing the operation (cost!).

```

φParams
┌───────────┴───────────┐
|   φBasicParams         |
|   op?   : Op           |
|   cost! : Money        |
└───────────┬───────────┘

```

Since all charges for this service depend only on the operation parameters, and not directly on the concrete state of the service, the definition of the φSSTariff framing schema given in the "User Manual" does not require further elaboration for the implementation.

The implemented service operations can then be brought together into a single definition as follows.

```

cBasicOps ≐
  (φSSTariff ^
    (cCreate   ^ φParams | op? = CreateOp   ) v
    (cRead    ^ φParams | op? = ReadOp    ) v
    (cStatus  ^ φParams | op? = StatusOp   ) v
    (cDestroy ^ φParams | op? = DestroyOp  ) v
    (cReplace ^ φParams | op? = ReplaceOp  ) v
    (cSetExpiry ^ φParams | op? = SetExpiryOp ) v
    (cGetIds  ^ φParams | op? = GetIdsOp   ) v
    (cGetCount ^ φParams | op? = GetCountOp ) v
    (cScavenge ^ φParams | op? = ScavengeOp ) v
    (cProfile ^ φParams | op? = ProfileOp  ))

```

7 Disk layout and caching

In this section, some refinements are made to the subsystems that have been presented so far. Since the permanent state of the service is to be stored on disk, it is necessary to describe the implementation of headers and bit-maps in terms of physical disk blocks. Together with the data-blocks, these then comprise the layout of information on the disk itself.

The specifications of the header and bit-map subsystems presented earlier can be considered as the intermediate abstract states and operations for which concrete implementations are now provided.

7.1 Header blocks

Typically, a number of block headers can be contained in each block stored on disk, so a header-block will consist of an array of block headers indexed by their position in the block.

$$\text{HeadersPerBlock} = \text{DiskBlockSize} \text{ div } \text{HeaderSize}$$

$$\text{HeaderBlockPos} \hat{=} 0.. \text{HeadersPerBlock}-1$$

$$\text{HeaderBlock} \hat{=} \text{HeaderBlockPos} \rightarrow \text{BlockHeader}$$

Header-blocks are given numbers from the set HeaderBlockNum , which must be large enough to allow headers to be stored for the maximum number of blocks in the service.

$$\text{HeaderBlockNum} : \text{F N}$$

$$\text{MaxBlocks} \leq \# \text{HeaderBlockNum} * \text{HeadersPerBlock}$$

Two functions indicate the header (identified by its header-block number and position within that header-block) associated with any particular numbered data-block. The exact definition of these functions depends on the chosen layout of blocks on the disk, and is not given here. However, each block must be associated with a different header.

$\begin{aligned} \text{HBN} &: \text{BlockNum} \rightarrow \text{HeaderBlockNum} \\ \text{HBP} &: \text{BlockNum} \rightarrow \text{HeaderBlockPos} \end{aligned}$ <hr/> $\forall \text{bn1, bn2:BlockNum} \mid \text{bn1} \neq \text{bn2} \cdot$ $(\text{HBN}(\text{bn1}) \neq \text{HBN}(\text{bn2})) \vee (\text{HBP}(\text{bn1}) \neq \text{HBP}(\text{bn2}))$
--

The headers subcomponent of the concrete service state is represented by the storage for the header-blocks, indexed by the corresponding header-block number.

$\text{HeaderBlocks} \quad \text{headerBlocks} : \text{HeaderBlockNum} \rightarrow \text{HeaderBlock}$
--

Initially, when the service is first started, all the header-blocks contain headers denoting unused blocks.

$\text{InitHeaderBlocks} \quad \text{HeaderBlocks}'$ <hr/> $\forall \text{bn:BlockNum} \cdot$ $\text{headerBlocks}'(\text{HBN}(\text{bn}))(\text{HBP}(\text{bn})).\text{used} = \text{False}$

The representation relation between the abstract Headers and the concrete HeaderBlocks is as follows.

$\text{RelHeaders} \quad \text{Headers}$ <hr/> HeaderBlocks <hr/> $\text{headers} = \lambda \text{bn:BlockNum} \cdot$ $\text{headerBlocks}(\text{HBN}(\text{bn}))(\text{HBP}(\text{bn}))$
--

A header buffer is also introduced as an additional state component, to hold a single header-block with a particular number, for the duration of a service operation only.

$\text{HeaderBuffer} \quad \text{hnum} : \text{HeaderBlockNum}$ $\text{hblock} : \text{HeaderBlock}$
--

The suboperations on the headers subsystem are implemented in terms of these new state components.

GetHeader

GetHeader (bn? : BlockNum; header! : Header)

```

cGetHeader _____
|
|  ΔHeaderBlocks
|  HeaderBuffer
|  bn?      : BlockNum
|  header!  : Header
|
|-----|
|
|  hnum     = HBN(bn?)
|  hblock   = headerBlocks(hnum)
|  header!  = hblock(HBP(bn?))
|
|_____|

```

PutHeader

PutHeader (bn? : BlockNum; header? : Header)

```

cPutHeader _____
|
|  ΔHeaderBlocks
|  HeaderBuffer
|  bn?      : BlockNum
|  header?  : Header
|  hblock'  : HeaderBlock
|
|-----|
|
|  hnum          = HBN(bn?)
|  hblock'       = hblock @ {HBP(bn?) ↦ header?}
|  headerBlocks' = headerBlocks @ {hnum ↦ hblock'}
|
|_____|

```

When putting a new header, it must have the same header-block number as the existing header-block in the buffer. This allows a new header-block to be formed by simply replacing the appropriate header component, leaving the rest of the header-block unchanged. This constraint is met in the service operation implementations, where each put is preceded by a get for the same block number.

7.2 Map blocks and cache

The bit-map showing block usage is stored within several physical blocks on the disk. Many bits are contained in each map-block, each bit being indexed by its position in the block.

$$\text{BitsPerBlock} = \text{DiskBlockSize} * \text{ByteSize}$$

$$\text{MapBlockPos} \hat{=} 0.. \text{BitsPerBlock} - 1$$

$$\text{MapBlock} \hat{=} \text{MapBlockPos} \rightarrow \text{Bit}$$

Map-blocks are given numbers from the set MapBlockNum , which must be large enough to allow bits to be stored for the maximum number of blocks in the service.

$$\text{MapBlockNum} : \mathbf{F} \mathbf{N}$$

$$\text{MaxBlocks} \leq * \text{MapBlockNum} * \text{BitsPerBlock}$$

Two functions indicate the bit (identified by its map-block number and position within that map-block) associated with any particular numbered data-block. The exact definition of these functions depends on the chosen layout of physical blocks on the disk, and is not given here. However, each block must be associated with a different bit in a map-block.

$$\text{MBN} : \text{BlockNum} \rightarrow \text{MapBlockNum}$$

$$\text{MBP} : \text{BlockNum} \rightarrow \text{MapBlockPos}$$

$$\forall \text{bn1}, \text{bn2} : \text{BlockNum} \mid \text{bn1} \neq \text{bn2} \cdot \\ (\text{MBN}(\text{bn1}) \neq \text{MBN}(\text{bn2})) \vee (\text{MBP}(\text{bn1}) \neq \text{MBP}(\text{bn2}))$$

The bit-map subcomponent of the concrete service state is represented by the storage for the map-blocks, indexed by the corresponding map-block number.

$$\text{MapBlocks} \xrightarrow{\hspace{10em}} \\ \boxed{\text{mapBlocks} : \text{MapBlockNum} \rightarrow \text{MapBlock}}$$

Initially, when the service is first started, all the map-blocks denote free blocks.

InitMapBlocks
MapBlocks'
\forall mbn:MapBlockNum • ran mapBlocks'(mbn) = {FreeBit}

In order to optimise the use of the bit-map, particularly to allow rapid identification of free blocks when allocating new blocks, the concrete service state includes a cache for a single map-block (it is called a cache here, rather than a buffer, since it persists between one service operation and the next). The cache holds the map-block and its map-block number.

MapCache
mnum : MapBlockNum mblock : MapBlock

Initially, the cache holds a copy of the lowest numbered map-block, which is all 'free'.

InitMapCache
MapCache'
mnum' = min MapBlockNum ran mblock' = {FreeBit}

The representation relation between the abstract BitMap and the concrete MapBlocks and MapCache is as follows. The cached map-block overrides the corresponding map-block stored on disk.

RelBitMap
BitMap MapBlocks MapCache
bitMap = λ bn:BlockNum • (mapBlocks \oplus {mnum \mapsto mblock})(MBN(bn))(MBP(bn))

The suboperations on the bit-map subsystem are implemented in terms of these new state components.

An additional suboperation is introduced for use in the following two suboperations which allocate or free the bit associated with a particular block. Given the number of a block whose corresponding bit is to be altered, this suboperation checks whether the bit is to be found in the cached map-block or not. If not, the cached map-block is flushed to disk and the relevant map-block obtained from disk. In either case, the the bit is set to the requested value in the finally cached map-block.

UpdateMap

UpdateMap (bn? : BlockNum; val? : Bit)

UpdateMap	
Δ MapBlocks	
Δ MapCache	
bn?	: BlockNum
val?	: Bit
newmblock	: MapBlock
<hr/>	
mnum' = MBN(bn?)	
mnum' = mnum \Rightarrow	
mapBlocks' = mapBlocks	
newmblock = mblock	
mnum' \neq mnum \Rightarrow	
mapBlocks' = mapBlocks \circ {mnum \mapsto mblock}	
newmblock = mapBlocks(mnum')	
mblock' = newmblock \circ {MBP(bn?) \mapsto val?}	

AllocBit

AllocBit (bn? : BlockNum)

cAllocBit	
UpdateMap [bn?, val/val?]	
bn?	: BlockNum
val	: Bit
<hr/>	
val	= UsedBit

FreeBit

FreeBit (bn? : BlockNum)

cFreeBit UpdateMap [bn?, val/val?] bn? : BlockNum val : Bit
val = <i>FreeBit</i>

A further operation is introduced to find the block number of a free block. The cached map-block is searched first for a free bit. Only if none is found will the map-blocks on disk be used. This could involve a scan of all the map-blocks at the next lower level of refinement of the implementation.

FindFreeBlock

FindFreeBlock (bn! : BlockNum)

cFindFreeBlock Δ MapBlocks Δ MapCache bn! : BlockNum
<i>FreeBit</i> \in ran mblock \Rightarrow MBN(bn!) = mnum mblock(MBP(bn!)) = <i>FreeBit</i> <i>FreeBit</i> \notin ran mblock \Rightarrow mapBlocks(MBN(bn!))(MBP(bn!)) = <i>FreeBit</i>

The corresponding error report, which indicates that there are no free blocks on the disk, also has to be refined in terms of the map-blocks and cache.

```

cNoSpace1
  ≡ cSS
  ≡ MapBlocks
  ≡ MapCache

  FreeBit ≡ ran mblock
  ∀ bn:BlockNum •
    FreeBit ≡ ran mapBlocks(MBN(bn))
  report! = NoSpaceReport

```

Clearly, only one scan of the disk would be required to either produce this error report, or to identify a free block. Therefore the schemas `FindFreeBlock` and `cNoSpace` would be implemented by the same code.

7.3 Disk layout

It is assumed that the Block Storage Service is to be implemented on a random access permanent storage device (such as a magnetic disk) which may be modelled as an array of fixed size blocks. Each disk block may be used to store a data-block, a header-block or a map-block.

```

DiskBlock ::= DiskData <<DataBlock>> |
             DiskHeader <<HeaderBlock>> |
             DiskMap <<MapBlock>>

```

The disk blocks are numbered consecutively up to the capacity of the disk.

```

DiskBlockNum ≡ 0..MaxDiskBlocks-1

```

```

Disk
  ┌───────────────────────────────────────────┐
  │ diskBlocks : DiskBlockNum → DiskBlock │
  └───────────────────────────────────────────┘

```

The number of blocks on the disk must be sufficient to hold all the required blocks of the service.

```

MaxDiskBlocks ≥ #BlockNum + #HeaderBlockNum + #MapBlockNum

```

The layout of the different kinds of blocks on the disk is given by the mappings from the specific block numbers to the disk block numbers. The layout is not specified in greater detail here, as it will depend on a particular disk design, but clearly the different kinds of block must occupy disjoint areas of the disk.

```

DLayout : BlockNum      >> DiskBlockNum
HLayout : HeaderBlockNum >> DiskBlockNum
MLayout : MapBlockNum   >> DiskBlockNum
-----
disjoint <ran DLayout, ran HLayout, ran MLayout>

```

The contents of the disk are simply the contents of the specific kinds of block placed in the appropriate locations according to the layout maps.

```

DiskContents -----
Disk
DataBlocks
HeaderBlocks
MapBlocks
-----
diskBlocks =
  DLayout-1 ; dataBlocks ; DiskData   U
  HLayout-1 ; headerBlocks ; DiskHeader U
  MLayout-1 ; mapBlocks ; DiskMap

```

8 Implementation correctness

It would be important to show that the implementation of the Block Storage Service as described in this manual correctly implements the view presented in the user manual.

In order to do this, each state refinement step expressed as an abstraction relation, whether of the whole service or of one of its subsystems, must be considered in turn.

For each one it must be shown that there exists a concrete state that represents each abstract state. For service initialisation, the concrete initial state must be shown to correspond to a valid abstract initial state. For each operation, it must be shown that the concrete operation may be applied whenever the abstract operation may be applied, and that it will then produce a result satisfying the abstract specification.

However, the implementations of the operations contained in this manual have not so far been proved correct in this respect. The manual must therefore be looked upon as an illustration of a style of implementation specification, rather than as containing a proven implementation design.

Appendix: Iterating schemas

If P is a schema which represents an operation on a state schema S (having undashed and dashed components representing the state before and after the operation), and B is a schema representing a predicate defined on S , iteration over P can be defined as follows.

Let

$$\begin{aligned} \cong S &\hat{=} S \wedge S' \mid \emptyset S = \emptyset S' \\ I_0 &\hat{=} \neg B \wedge \cong S \\ I_{i+1} &\hat{=} (B \wedge P) \mid I_i \quad \forall i:N \end{aligned}$$

then

$$P \text{ while } B \hat{=} I_0 \vee I_1 \vee I_2 \vee \dots$$

Chapter 3

Discussion and Experience

- 1 Introduction
- 2 History of development
 - 2.1 Original design
 - 2.2 Implementation
 - 2.3 Implementor manual
- 3 Design of the user interface
 - 3.1 Limited life
 - 3.2 Immutability
 - 3.3 Tags
 - 3.4 Key-linked operations
 - 3.5 Lifetime vs. expiry
 - 3.6 Manager operations
 - 3.7 Scavenging
 - 3.8 Structured parameters
- 4 Format of the User Manual
 - 4.1 Errors
 - 4.2 Common framework
- 5 Design of the implementation
 - 5.1 Lack of concurrency
 - 5.2 Fault handling
 - 5.3 Disk layout
 - 5.4 Consistency and crash recovery
- 6 Format of the Implementor Manual
 - 6.1 Initial versions
 - 6.2 Current version
 - 6.3 Correctness concerns

1 Introduction

The design and documentation of the Block Storage Service has been developed in stages over the duration of the project. The history of this development is summarised in the next section.

The subsequent sections look at the design choices and the documentation from the point of view of both the user and the implementor of the service, corresponding to the two manuals contained in the previous chapters. Alternative design choices are discussed for each of the two levels of abstraction, and some comments made on the way in which the manuals have been presented.

This chapter concentrates on the experience gained from the specification and implementation of the Block Storage Service in particular. Some general improvements in manual style, and the introduction of the Common Service Framework to provide the definition of common service characteristics, are both discussed in "The Specification of Network Services" [8].

2 History of development

2.1 Original design

The original design of the user interface to the Block Storage Service was developed, and specified in Z, by Carroll Morgan and Roger Gimson. This led to the production of the first User Manual for which many of the conventions of presentation shown here were first devised. It is this design which was presented as part of a monograph at the end of the first stage of the project [1].

2.2 Implementation

Having designed the user's view of the service, and produced a User Manual, an implementation was designed and coded by Carroll Morgan without further use of formal techniques. The objective at this stage was to get something working so that the feasibility of the user interface could be assessed. In any case, no formal refinements of significant size had been undertaken at that stage in the development of the Z notation.

The implementation (written in Modula-2 running on an LSI-11) was found to be adequate, and is still essentially unchanged. The service provides data storage facilities for spooled laser printer output, and a certain amount of backup storage, for members of the Programming Research Group to this day.

2.3 Implementor manual

In the second part of the project, it was decided to produce an Implementor Manual for the Block Storage Service that reflected the existing implementation. Though this is not the recommended methodology (after all, one of the main objectives of the use of formal specification methods is to more clearly express design choices at an abstract level before producing any code), it allowed implementation design decisions to be assumed while the presentation and structure of the specification in the manual were considered.

The Implementor Manual has been through two earlier versions before the form presented in the previous chapter was evolved. Jonathan Bowen produced the first version, which was then rewritten and extended by Roger Gimson. Changes have largely been motivated by the wish to structure the design into separately understandable subsystems, so giving the implementor firm guidelines to the structure of the final code.

Though the design essentially reflects the structure of the existing implementation, it was found that the suboperations on the components of the service state didn't necessarily correspond to routines in the existing code. If time had permitted, it would have been an interesting exercise to rewrite the code to conform to the manual.

3 Design of the user interface

The design of a low-level data storage service was chosen in order to keep the complexity of the service under control, while providing a basis for the implementation of higher-level facilities. Such a separation is not novel in itself (see, for example, the Amoeba system [9]), though the design turned out to incorporate some unusual characteristics.

3.1 Limited life

The initial design choices were greatly influenced by a model of a dry-cleaning service, originally proposed by Tony Hoare as a suitable study of an existing human-oriented service.

The idea of enforcing a fixed lifetime on the data arose from that study. A dry-cleaning service will dispose of any clothes that have been left for cleaning but not been claimed after a suitably long period. By this analogy we mitigated the fear that users would object if their data suddenly vanished at some point in the future.

Clearly the storage media implementing any particular service will not last forever — but users are conventionally happier thinking that their data will remain there indefinitely until they explicitly force change.

In practice, lifetimes have been used in two distinct ways. For temporary data, such as spooled printer output, a short lifetime is used (of perhaps one or two days), so that data will normally expire rather than be explicitly destroyed. For permanent data, such as backup storage, a medium or long lifetime is used (from a few months to a year or more); occasionally an archive program will be used by a client to explicitly destroy data which is no longer required, and extend the expiry time of data to be retained.

3.2 Immutability

The initial design also embodied the idea that stored blocks were immutable. Any given block identifier could only ever refer to the 'same' stored data. There is no operation which can change the data associated with a particular identifier.

Particularly in a shared service, the property of immutability is very valuable since it allows a user to be sure that an identifier that they hold can only refer to a particular data item, irrespective of operations being performed by other users. It presents the same kinds of advantages and disadvantages as in the manipulation of data structures in purely functional programming languages.

In some applications with tree-structured data, such as directories of files, an underlying immutable implementation imposes the condition that any change to a leaf of the structure also changes the complete path back to the root. For balanced structures, this introduces a penalty of at worst logarithmic complexity. It may also have the disadvantage of requiring all references to the data to be channelled through

the single root node.

However, immutability does, for example, provide a natural way to do check-pointing. Assuming the components of the structure are not destroyed (or do not expire), a snapshot of a complete tree structure may be held as a single reference to the root.

To ensure strict immutability, and if the expiry time of a block is considered as part of its value, the service operation which changes the expiry time should also change the identity of the block. This was not done in the implemented system, mainly from a wish to be able to change the lifetime of a tree-structured object, like a file, without rewriting all its non-leaf nodes.

3.3 Tags

At one stage in the initial design, the 'tag' part of the data in a block (that part which extends the size of a block to be slightly larger than a conventional disk block) was distinguished in the user interface, so that its value was provided separately in a block creation operation, and could be returned as part of a status operation. This distinction was dropped as being too 'implementation-inspired'.

Later discussions about implementing other services on top of the Block Storage Service raised the possibility of making the tag mutable. It could then be used, for example, for storing a reference count to the block without having to change the associated block id. This would be a relatively straightforward change to both the specification and implementation of the service, though it would introduce the need for further operations to set and get the tag field.

3.4 Key-linked operations

One place where implementation issues do intrude more than they might is in the 'key-linked' operations. These arise from the wish to make each service operation correspond directly to a single network procedure call, which for practical reasons is limited in duration and size of parameters. Since these operations are required to return sets or sequences of potentially large size, and could take correspondingly long to execute, they each return only a part of the desired set or sequence.

This partitioning could be hidden from the user at a higher level by defining a single operation which would be implemented by the appropriate sequence of key-linked operations, and which would construct the whole of the resultant parameter. However,

it might be misleading to define this single higher-level operation as an atomic operation on the state of the service; at least the current formulation correctly describes the effect of other user operations interleaved between the components of a key-linked sequence.

3.5 Lifetime vs. expiry

The user interface design presented in this monograph differs from the first phase design in some small but significant ways.

The definition of the lifetime of a block as an interval from the creation time has been replaced by explicit definition of the expiry time. The latter makes it possible to ensure that a set of blocks all have the same expiry time, whereas in the former this could vary according to the time of invocation of an operation. This means that, when implementing a higher-level data structure from blocks, there can be a simple invariant that all the constituent blocks will exist for as long as the higher-level structure exists.

3.6 Manager operations

Another difference from the initial design is that the scavenge operation has been made an explicit manager operation, rather than an asynchronous internal operation of the service. This corresponds to a change made to the implementation which allowed the scavenging to be invoked as an explicit operation.

The profile operation was also added as another manager operation. The normal service user does not need to be aware of these operations. However, they do form a legitimate part of the user interface, if only for the special user who is the manager.

3.7 Scavenging

There is still a debatable point of design concerning scavenging. As the service is presented here, a block is still accessible until it is scavenged, even though it may have passed its expiry time.

In the first version of the design, with scavenge as an asynchronous internal operation, the implementor was given some freedom of choice. The scavenge could be considered to occur immediately before an access was attempted, resulting in no access to such blocks, or to occur sometime later, meaning such blocks might be accessible.

There may be a good case for forcing blocks to disappear as soon as they reach their expiry time, so that, for example, it is known that if one block in a higher-level data structure has expired they all must have expired. This can be achieved in the current design by simply including, as part of the test in the error schema `NoSuchBlock`, a check as to whether the block is past its expiry time.

3.8 Structured parameters

Some service operations return structured data items as results. For example, `GetIds` returns a set of block ids, while `Profile` returns a sequence of block information. In the latter case, a set cannot be used because each piece of block information is not necessarily unique (it contains owner and creation and expiry times, but not the unique block id) so replicated entries are significant. A bag might have been a more accurate abstract specification for this parameter, but would have had a less obvious concrete representation.

As it turned out, the implementation described in the foregoing *Implementor Manual* would have guaranteed uniqueness of each piece of block information, since it relies on each block having a different creation time. However, building this fact into the *User Manual* would unnecessarily limit the choice of implementations.

The representation of structured parameters is briefly discussed in [8].

4 Format of the User Manual

The overall design of the *User Manual* follows that of a typical manual for a library of system calls, with an introductory overview of the system followed by detailed descriptions of each of the operations that can be invoked on the system. The use of a formal notation ensures that the user's view of the system is made much more explicit than is usual in informal manuals, though without introducing unnecessary implementation detail.

4.1 Errors

The specification of behaviour under error conditions is also covered in detail, though not at the expense of cluttering the description of the successful behaviour of an

operation. The specification of error conditions, formalised in the 'Reports' section of each operation description, has changed between the first version of the manual and that presented here. Schema overriding is now used to explicitly define the order in which errors may be detected. In some cases, such ordering is essential. For example, it is necessary to check whether a block exists before its ownership can be checked.

4.2 Common framework

The introduction of the "Common Service Framework" has also made a difference to the presentation. Parameters which can be considered implicit to every operation have been separated, including identifications of the client, operation and service involved in a particular call. The combination of the individual operation specifications into an overall specification for the whole service, including subsystems common to other services, now forms the final part of the manual.

5 Design of the implementation

The implementation described here is simple, but it has been found to be adequate for the straightforward applications it has supported over the three years it has been in use.

5.1 Lack of concurrency

One major simplification in the implementation is the lack of provision for concurrent execution of service operations. Though the user interface is specified as if each operation were atomic, this does not necessarily force the implementation to be sequential, provided that the effect of executing two service operations 'in parallel' is equivalent, as far as the user is concerned, to executing first one then the other (in either order).

A sequential implementation means that the time taken to execute any one operation should be strictly limited to ensure that other users are not kept waiting for too long. For example, this is one reason for defining a limit on the number of blocks scanned in key-linked operations.

5.2 Fault handling

Another simplification in the implementation described here is that no account has been taken of potential faults in the underlying hardware, such as bad disk blocks. Faults that are unrecoverable (for example, cannot be cured by re-reading the disk block) are allowed for in the abstract service specification by the catch-all error report `ServiceError`. In this event, the state of the service is specified to remain unchanged.

An implementation which caters for such errors must ensure that any changes in the concrete state made prior to the detection of the fault are recoverable, or at least do not lead to an inconsistent concrete state or changed abstract state. In practice, this can often be achieved through appropriate choice of the order of execution of suboperations, and simply abandoning further calls on detection of the fault.

5.3 Disk layout

The implementation does not specify in detail how the various blocks of information should be laid out on the disk. This would depend on the characteristics of a particular disk drive. However, it will generally be a good idea, in order to reduce disk arm movement, to place the data, header and bitmap disk blocks associated with a stored service block in the same area of the disk.

In the actual implementation produced as part of the project, the disk format consisted of 32 blocks per track, with 16 tracks per cylinder. One disk block could hold up to 8 headers, or 4096 bits of a bit-map. The chosen layout allocated the first 4 blocks of each track to hold headers relating to the remaining 28 blocks used for data. Every 8 cylinders, the last block of a track was used for a bit-map (instead of data) relating to all the service blocks stored on those cylinders.

5.4 Consistency and crash recovery

The criteria given for consistency within the implementation, showing how the bit-map and count information should be consistent with the header information, would imply the restoration of this consistency after any service crash.

Crashes are not modelled explicitly in the manuals. They can be considered as periods during which all service calls will return an error report. There is no allowance for the loss of information which might have occurred during the crash, the service state being defined to remain unchanged for such errors. On rebooting the service after the crash,

the specified consistency constraints are assumed to be re-established.

In the actual implementation, the count information is held in memory, and so must be recomputed by scanning the complete disk. (It is assumed that crashes will be infrequent, as has been the case in practice.)

The bitmaps stored on disk should also be recomputed while scanning the headers. In fact the actual implementation used a somewhat looser notion of consistency than specified here. For a specific service block, the header might be marked 'used', but the bitmap indicates 'free', in which case the header is believed and the bitmap is corrected on an attempt to access that block. Alternatively, the header might be marked 'free', but the bitmap indicates 'used', in which case the block becomes temporarily unavailable for further allocation. A utility program can be executed occasionally to restore full consistency and recover such unusable resources.

This extra complexity was intended to allow faster rebooting after a crash (the count information was deemed to be unavailable, and a scan of the whole disk avoided). In hindsight, crashes are so infrequent that the simpler consistency criteria specified in the Implementor Manual would have been adequate.

6 Format of the Implementor Manual

The Implementor Manual went through three distinct forms in an attempt to provide a sufficiently readable presentation.

6.1 Initial versions

In the first version, the concrete state was developed explicitly as a number of separate refinement steps from the abstract state given in the User Manual. The operations, however, were defined as monolithic schemas on the concrete state alone, which made them rather large and difficult to understand.

In the second version, the concrete state was introduced directly as a number of subcomponents, including those relating to disk layout and caching. The operations were composed from suboperations involving the separate state subcomponents. Only towards the end of the manual was there a description of how the concrete state related to the abstract state. In this version, the number of subcomponents of the

concrete state made it difficult to remember what was affected by a particular suboperation forming part of a service operation description.

6.2 Current version

In the third form, as presented here, a balance was struck between the two previous versions. Two levels of abstraction were used. The concrete state is introduced as a few subsystems at an intermediate level of abstraction (such as those for headers and a single large bit-map), and related to the abstract state. The service operations are composed from suboperations involving these intermediate subsystems. Then the intermediate subsystems are refined one stage further to include details of disk layout and bit-map blocks and caching.

The refinement of each intermediate subsystem can be understood on its own, without reference to the overall behaviour of the service. In this sense, the format of the presentation more closely follows the use of abstraction and abstract data types (corresponding to the subsystems) in conventional system design.

6.3 Correctness concerns

However, none of the versions of the implementor manual have been oriented towards the requirements for proving the correctness of the implementation. Indeed, there has not been sufficient time within the project to attempt such a proof for a service of even this moderate complexity (though a simpler one has been completed [10]).

It therefore remains an open question as to whether the strictures necessary to allow a proof to be completed would enforce a further change in implementation specification style. It is quite possible that the proof would be easier to carry out if associated with the application of smaller refinement steps. However, it is not clear whether an implementor would benefit from seeing these details in the manual.

Acknowledgements

Carroll Morgan played a large part in the initial design, and was wholly responsible for the first implementation, of the Block Storage Service. Jonathan Bowen produced an initial version of the Implementor Manual, a command script for producing the index and the glossary. The other members of the Distributed Computing Software Project, Tim Gleeson and Stig Topp-Jørgensen, have also influenced the presentation of the manuals. All the above, plus Bernard Sufrin, have made valuable comments on earlier drafts of this monograph. Thanks also to those working on the development of Z in the several related projects at the Programming Research Group.

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

References

1. Gimson, R.B., Morgan, C.C. "The Distributed Computing Software Project", Technical Monograph PRG-50, *Programming Research Group, Oxford University* (1985).
2. Sufrin, B.A. (editor) "Z Handbook", Draft 1.1, *Programming Research Group, Oxford University* (1986).
3. Spivey, J.M. "Understanding Z: A Specification Language and its Formal Semantics", D.Phil. Thesis, *Programming Research Group, Oxford University* (1986).
4. Spivey, J.M. "The Z Library - A Reference Manual", *Programming Research Group, Oxford University* (1986).
5. Woodcock, J. "Structuring Specifications - Notes on the Schema Notation", *Programming Research Group, Oxford University* (1986).
6. King, S., Sørensen, I.H., Woodcock, J. "Z: Concrete and Abstract Syntaxes", Version 1.0, *Programming Research Group, Oxford University* (1987).
7. Hayes, I.J. (editor) "Specification Case Studies", *Prentice-Hall International Series in Computer Science* (1987).
8. Bowen, J.P., Gimson, R.B., Topp-Jørgensen, S. "The Specification of Network Services", Technical Monograph PRG-61, *Programming Research Group, Oxford University* (1987).
9. Mullender, S.J. "Principles of Distributed Operating System Design", PhD Thesis, *CWI Amsterdam* (1985).
10. Topp-Jørgensen, S. "Reservation Service: Implementation Correctness Proofs", DCS Project working paper, *Programming Research Group, Oxford University* (1987).

Appendix A

Index of formal definitions

The following index lists the page numbers on which each formal name is defined in the text. Those names which are defined twice correspond to duplicated entries in the User and Implementor Manuals. Names which have a special symbol (Δ , Φ , \equiv , c) as a prefix are listed after the corresponding base name.

AllocBit	44	EndInfoIteration	74
cAllocBit	81	EndScan	55
cBadId	56	EndScavengeIteration	71
BadKey	16	ErrorTariff	31
cBadKey	57	FetchCount	46
cBasicOps	75	FindFreeBlock	45
ϕBasicParams	11, 50	cFindFreeBlock	82
Bit	39	FreeBit	45
BitMap	44	cFreeBit	82
≡BitMap	50	GetAttributes	52
BitMapConsistency	49	GetBlockId	66
Block	9, 37	GetBlockInfo	73
ϕBlock	12	GetBlockNum	48
BlockCount	12, 38	GetCount	26
BlockData	9, 37	cGetCount	68
BlockIdParts	47	GetCount _{success}	26
BlockIdSet	12, 38	cGetCount _{success}	68
BlockInfo	9, 37	GetData	41
BlockInfoSeq	12, 38	GetHeader	43
Byte	39	cGetHeader	78
CheckBlock	71	GetIds	25
CountConsistency	49	cGetIds	67
Counts	45	GetIds _{success}	25
≡Counts	50	cGetIds _{success}	67
Create	19	Header	42
cCreate	60	HeaderBlock	76
Create _{success}	19	HeaderBlockPos	76
cCreate _{success}	60	HeaderBlocks	77
DataBlock	40	HeaderBuffer	77
DataBlocks	41	Headers	42
≡DataBlocks	50	IncCount	46
DecCount	46	InitBitMap	44
Destroy	22	InitCounts	46
cDestroy	63	InitDataBlocks	41
Destroy _{success}	22	InitHeaderBlocks	77
cDestroy _{success}	63	InitHeaders	43
Disk	83	InitMapBlocks	80
DiskBlock	83	InitMapCache	80
DiskBlockNum	83	InitSS	10, 38
DiskContents	84	cInitSS	51
EndIdsIteration	67	InitSSState	32

φKey	14	cReplace _{success}	64
MapBlock	79	SS	10, 37
MapBlockPos	79	ΔSS	11
MapBlocks	79	≡SS	11
MapCache	80	cSS	49
cMismatchedId	56	ΔcSS	50
NewAttributes	53	≡cSS	50
φNewBlock	12	SSAllOps	33
NewBlockId	48	SSBasicOps	32
NoSpace	16	SSOps	33
cNoSpace	56	SSServiceOps	32
cNoSpace ₁	83	SSState	32
NoSuchBlock	15	SSTeriff	31
cNoSuchBlock	56	φSSTariff	31
NotExpiredBlock	71	Scan	54
NotManager	16	ΔScan	54
cNotManager	57	ΔScanIds	66
NotOwner	16	ΔScanInfo	73
cNotOwner	57	ΔScanScavenge	70
φParams	30, 75	Scavenge	28
Profile	29	cScavenge	72
cProfile	74	ScavengeBlock	70
Profile _{success}	29	Scavenge _{success}	28
cProfile _{success}	74	cScavenge _{success}	72
PutData	41	SetExpiry	24
PutHeader	43	cSetExpiry	65
cPutHeader	78	SetExpiry _{success}	24
Read	20	cSetExpiry _{success}	65
cRead	61	StartScan	54
Read _{success}	20	Status	21
cRead _{success}	61	cStatus	62
RelBitmap	80	Status _{success}	21
RelHeaders	77	cStatus _{success}	62
RelSS	51	Success	15
Replace	23	cSuccess	55
cReplace	64	Teg	42
Replace _{success}	23	UpdateMap	81

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 [6].

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

2. Logic.

Let P, Q be predicates and D declarations.

- $\sim P$ Negation: "not P ".
- $P \wedge Q$ Conjunction: " P and Q ".
- $P \vee Q$ Disjunction: " P or Q ".
 $\hat{=} \sim(\sim P \wedge \sim Q)$.
- $P \Rightarrow Q$ Implication: " P implies Q " or "if P then Q ": $\hat{=} \sim P \vee Q$.
- $P \Leftrightarrow Q$ Equivalence: " P is logically equivalent to Q ":
 $\hat{=} (P \Rightarrow Q) \wedge (Q \Rightarrow P)$.
- true Logical constant.
- false $\hat{=} \sim \text{true}$
- $\forall D \cdot P$ Universal quantification:
"for all D, P holds".
- $\exists D \cdot P$ Existential quantification:
"there exists D such that P ".
- $\exists_1 D \cdot P$ Unique existence: "there exists a unique D such that P ".
- $\forall D | P \cdot Q \hat{=} (\forall D \cdot P \Rightarrow Q)$.
- $\exists D | P \cdot Q \hat{=} (\exists D \cdot P \wedge Q)$.

- $P \text{ where } D | Q$ Where clause:
 $\hat{=} \exists D | Q \cdot P$
- $P \text{ where } x_1 \hat{=} t_1; \dots ; x_n \hat{=} t_n$ Where clause:
 P holds, with the syntactic definition(s) defined locally.
- $D \vdash P$ Theorem: $\hat{=} \vdash \forall D \cdot P$.

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: $\hat{=} \sim(t_1 = t_2)$.
- $t \in S$ Set membership: " t is an element of S ".
- $t \notin S$ Non-membership: $\hat{=} \sim(t \in S)$.
- \emptyset Empty set: $\hat{=} \{x : X | \text{false}\}$.
- $S \subseteq T$ Set inclusion:
 $\hat{=} (\forall x : S \cdot x \in T)$.
- $S \subset T$ Strict set inclusion:
 $\hat{=} S \subseteq T \wedge S \neq T$.
- $\{t_1, t_2, \dots, t_n\}$ The set containing t_1, t_2, \dots and t_n .
- $\{D | P \cdot t\}$ The set of t 's such that given the declarations D, P holds.
- $\{D | P\}$ Given $D \hat{=} x_1 : T_1; \dots ; x_n : T_n$,
 $\hat{=} \{D | P \cdot \{x_1, \dots, x_n\}\}$.
- $\{D \cdot t\} \hat{=} \{D | \text{true} \cdot t\}$.
- (t_1, t_2, \dots, t_n) Ordered n -tuple of t_1, t_2, \dots and t_n .
- $T_1 \times T_2 \times \dots \times T_n$ Cartesian product:
the set of all n -tuples such that the i th component is of type T_i .
- $P S$ Powerset: the set of all subsets of S .
- $P_1 S$ Non-empty powerset:
 $\hat{=} P S \setminus \{\emptyset\}$.
- $F S$ Set of finite subsets of S :
 $\hat{=} \{T : P S | T \text{ is finite}\}$.
- $F_1 S$ Non-empty finite set:
 $\hat{=} F S \setminus \{\emptyset\}$.

$S \cap T$ Set intersection: given $S, T: \mathbf{P} X$,
 $\hat{=} \{x: X \mid x \in S \wedge x \in T\}$.

$S \cup T$ Set union: given $S, T: \mathbf{P} X$,
 $\hat{=} \{x: X \mid x \in S \vee x \in T\}$.

$S \setminus T$ Set difference: given $S, T: \mathbf{P} X$,
 $\hat{=} \{x: X \mid x \in S \wedge x \notin T\}$.

$\cap SS$ Distributed set intersection:
 given $SS: \mathbf{P} (\mathbf{P} X)$,
 $\hat{=} \{x: X \mid (\forall S: SS \cdot x \in S)\}$.

$\cup SS$ Distributed set union:
 given $SS: \mathbf{P} (\mathbf{P} X)$,
 $\hat{=} \{x: X \mid (\exists S: SS \cdot x \in S)\}$.

$\#S$ Size (number of distinct
 elements) of a finite set.

$\mu D \mid P \cdot t$ Arbitrary choice from the
 set $\{D \mid P \cdot t\}$.

$\mu D \cdot t \hat{=} \mu D \mid \text{true} \cdot t$

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 :
 $\hat{=} \mathbf{P} (X \times Y)$.

$x R y$ x is related by R to y :
 $\hat{=} \langle x, y \rangle \in R$. (R is often
 underlined for clarity.)

$x \mapsto y$ Maplet: $\hat{=} \langle x, y \rangle$.

$\text{dom } R$ The domain of a relation:
 $\hat{=} \{x: X \mid \exists y: Y \cdot x R y\}$.

$\text{ran } R$ The range of a relation:
 $\hat{=} \{y: Y \mid \exists x: X \cdot x R y\}$.

$R_1 \# R_2$ Forward relational composition:
 given $R_1: X \leftrightarrow Y$; $R_2: Y \leftrightarrow Z$,
 $\hat{=} \{x: X; z: Z \mid \exists y: Y \cdot$
 $x R_1 y \wedge y R_2 z\}$.

$R_1 \circ R_2$ Relational composition:
 $\hat{=} R_2 \# R_1$.

R^{-1} Inverse of relation R :
 $\hat{=} \{y: Y; x: X \mid x R y\}$.

$\text{id } X$ Identity function on the set X :
 $\hat{=} \{x: X \cdot x \mapsto x\}$.

R^k The relation R composed with
 itself k times: given $R: X \leftrightarrow X$,
 $R^0 \hat{=} \text{id } X$, $R^{k+1} \hat{=} R^k \circ R$.

R^* Reflexive transitive closure:
 $\hat{=} \cup \{n: \mathbf{N} \cdot R^n\}$.

R^+ Non-reflexive transitive closure:
 $\hat{=} \cup \{n: \mathbf{N}_1 \cdot R^n\}$.

$R(S)$ Relational image: given $S: \mathbf{P} X$,
 $\hat{=} \{y: Y \mid \exists x: S \cdot x R y\}$.

$S \triangleleft R$ Domain restriction to S :
 given $S: \mathbf{P} X$,
 $\hat{=} \{x: X; y: Y \mid x \in S \wedge x R y\}$.

$S \triangleleft R$ Domain subtraction:
 given $S: \mathbf{P} X$,
 $\hat{=} (X \setminus S) \triangleleft R$.

$R \triangleright T$ Range restriction to T :
 given $T: \mathbf{P} Y$,
 $\hat{=} \{x: X; y: Y \mid x R y \wedge y \in T\}$.

$R \triangleright T$ Range subtraction of T :
 given $T: \mathbf{P} Y$,
 $\hat{=} R \triangleright (Y \setminus T)$.

$_R$ 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.

$X \twoheadrightarrow Y$ The set of partial functions from
 X to Y :
 $\hat{=} \{f: X \leftrightarrow Y \mid \forall x: \text{dom } f \cdot$
 $(\exists! y: Y \cdot x f y)\}$.

$X \rightarrow Y$ The set of total functions from
 X to Y :
 $\hat{=} \{f: X \twoheadrightarrow Y \mid \text{dom } f = X\}$.

$X \rightsquigarrow Y$ The set of partial injective (one-to-one) functions from X to Y :
 $\hat{=} \{f : X \rightarrow Y \mid \forall y : \text{ran } f \cdot (\exists_1 x : X \cdot f x = y)\}$.

$X \rightarrow Y$ The set of total injective functions from X to Y :
 $\hat{=} (X \rightsquigarrow Y) \cap (X \rightarrow Y)$.

$X \twoheadrightarrow Y$ The set of partial surjective functions from X to Y :
 $\hat{=} \{f : X \twoheadrightarrow Y \mid \text{ran } f = Y\}$.

$X \twoheadrightarrow Y$ The set of total surjective functions from X to Y :
 $\hat{=} (X \twoheadrightarrow Y) \cap (X \rightarrow Y)$.

$X \xrightarrow{\sim} Y$ The set of total bijective (injective and surjective) functions from X to Y :
 $\hat{=} (X \twoheadrightarrow Y) \cap (X \rightarrow Y)$.

$X \dashrightarrow Y$ The set of finite partial functions from X to Y :
 $\hat{=} \{f : X \dashrightarrow Y \mid f \in \mathbf{F} (X \times Y)\}$.

$\rightarrow \rightsquigarrow \twoheadrightarrow \dashrightarrow$ Partial functions.

$\rightarrow \rightarrow \rightarrow \rightarrow$ Total functions.

$\rightarrow \twoheadrightarrow \twoheadrightarrow \dashrightarrow$ Finite functions.

$f_1 \circ f_2$ Functional overriding: given $f_1, f_2 : X \rightarrow Y$,
 $\hat{=} (\text{dom } f_2 \triangleleft f_1) \cup f_2$.

$f _$ 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 .

$f(t) \hat{=} f t$.

$\lambda D \mid P \cdot t$ Lambda-abstraction: the function that, given an argument x of type X such that P holds, the result is t . Given $D \hat{=} x_1 : T_1 ; \dots ; x_n : T_n$,
 $\hat{=} \{D \mid P \cdot (x_1, \dots, x_n) \mapsto t\}$.

$\lambda D \cdot t \hat{=} \lambda D \mid \text{true} \cdot t$

6. Numbers.

Let m, n be natural numbers.

\mathbf{N} The set of natural numbers (non-negative integers).

\mathbf{N}_1 The set of strictly positive natural numbers: $\hat{=} \mathbf{N} \setminus \{0\}$.

\mathbf{Z} The set of integers (positive, zero and negative).

$\text{succ } n$ Successive ascending natural number.

$\text{pred } n$ Previous descending natural number: $\hat{=} \text{succ}^{-1} n$.

$m + n$ Addition: $\hat{=} \text{succ}^n m$.

$m - n$ Subtraction: $\hat{=} \text{pred}^n m$.

$m * n$ Multiplication: $\hat{=} (_ + m)^n 0$.

$m \text{ div } n$ Integer division.

$m \text{ mod } n$ Modulo arithmetic.

m^n Exponentiation: $\hat{=} (_ * m)^n 1$.

$m \leq n$ Less than or equal, Ordering: $_ \leq _ \hat{=} \text{succ}^*$.

$m < n$ Less than, Strict ordering: $\hat{=} m \leq n \wedge m \neq n$.

$m \geq n$ Greater than or equal: $\hat{=} n \leq m$.

$m > n$ Greater than: $\hat{=} n < m$.

$m..n$ Range: $\hat{=} \{k : \mathbf{N} \mid m \leq k \wedge k \leq n\}$.

$\text{min } S$ Minimum of a finite set; for $S : \mathbf{F}_1 \mathbf{N}$, $\text{min } S \in S \wedge (\forall x : S \cdot x \geq \text{min } S)$.

$\text{max } S$ Maximum of a finite set; for $S : \mathbf{F}_1 \mathbf{N}$, $\text{max } S \in S \wedge (\forall x : S \cdot x \leq \text{max } S)$.

7. Orders.

$\text{partial_order } X$

The set of partial orders on X :

$\hat{=} \{R : X \leftrightarrow X \mid \forall x, y, z : X \cdot x R x \wedge x R y \wedge y R x \Rightarrow x = y \wedge x R y \wedge y R z \Rightarrow x R z\}$.

total_order X

The set of total orders on X:
 $\hat{=} \{R: \text{partial_order} \mid \forall x, y: X \cdot$
 $x R y \vee y R x\}$.

monotonic $X <_X$

The set of functions from X to X that are monotonic with respect to the order $<_X$ on X:
 $\hat{=} \{f: X \rightarrow X \mid \forall x, y: X \cdot$
 $x <_X y \Rightarrow f(x) <_X f(y)\}$.

8. Sequences.

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

seq X

The set of sequences whose elements are drawn from X:
 $\hat{=} \{A: \mathbb{N} \rightarrow X \mid$
 $\text{dom } A = 1.. \#A\}$.

$\langle \rangle$

The empty sequence \emptyset .

seq₁ X

The set of non-empty sequences:
 $\hat{=} \text{seq } X \setminus \{\langle \rangle\}$

$\langle a_1, \dots, a_n \rangle$

$\hat{=} \{1 \mapsto a_1, \dots, n \mapsto a_n\}$.

$\langle a_1, \dots, a_n \rangle \hat{\sim} \langle b_1, \dots, b_m \rangle$

Concatenation:
 $\hat{=} \langle a_1, \dots, a_n, b_1, \dots, b_m \rangle,$
 $\langle \rangle \hat{\sim} A = A \hat{\sim} \langle \rangle = A.$

head A

The first element of a non-empty sequence:
 $A \neq \langle \rangle \Rightarrow \text{head } A = A(1).$

last A

The final element of a non-empty sequence:
 $A \neq \langle \rangle \Rightarrow \text{last } A = A(\#A).$

tail A

All but the head of a sequence:
 $\text{tail}(\langle x \rangle \hat{\sim} A) = A.$

front A

All but the last of a sequence:
 $\text{front}(A \hat{\sim} \langle x \rangle) = A.$

rev $\langle a_1, a_2, \dots, a_n \rangle$

Reverse:
 $\hat{=} \langle a_n, \dots, a_2, a_1 \rangle,$
 $\text{rev } \langle \rangle = \langle \rangle.$

$\hat{\sim} / AA$

Distributed concatenation:

given $AA : \text{seq}(\text{seq}(X)),$
 $\hat{=} AA(1) \hat{\sim} \dots \hat{\sim} AA(\#AA),$
 $\hat{\sim} / \langle \rangle = \langle \rangle.$

\ddagger / AR

Distributed relational composition:
 given $AR : \text{seq}(X \leftrightarrow X),$
 $\hat{=} AR(1) \ddagger \dots \ddagger AR(\#AR),$
 $\ddagger / \langle \rangle = \text{id } X.$

\oplus / AR

Distributed overriding:
 given $A : \text{seq}(X \rightarrow Y),$
 $\hat{=} AR(1) \oplus \dots \oplus AR(\#AR),$
 $\oplus / \langle \rangle = \emptyset.$

squash f

Convert a finite function, $f: \mathbb{N} \rightarrow X,$ into a sequence by squashing its domain. That is, $\text{squash } \emptyset = \langle \rangle,$ and if $f \neq \emptyset$ then $\text{squash } f =$
 $\langle f(i) \rangle \hat{\sim} \text{squash}(\{i \nmid f\})$
 where $i = \min(\text{dom } f).$

$S \downarrow A$

Index restriction:
 $\hat{=} \text{squash}(S \downarrow A).$

$A \uparrow T$

Sequence restriction:
 $\hat{=} \text{squash}(A \uparrow T).$

disjoint AS

Pairwise disjoint:
 given $AS : \text{seq}(\mathbb{P} X),$
 $\hat{=} (\forall i, j : \text{dom } AS \cdot i \neq j$
 $\Rightarrow AS(i) \cap AS(j) = \emptyset).$

AS partitions S

$\hat{=} \text{disjoint } AS \wedge$
 $\bigcup \text{ran } AS = S.$

A in B

Contiguous subsequence:
 $\hat{=} (\exists C, D : \text{seq } X \cdot$
 $C \hat{\sim} A \hat{\sim} D = B).$

9. Bags.

bag X

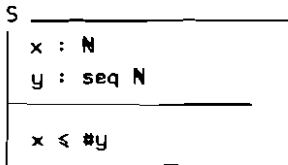
The set of bags whose elements are drawn from X: $\hat{=} X \rightarrow \mathbb{N}_1$

items s

The bag of items contained in the sequence s: $\hat{=} \{x : \text{ran } s \cdot$
 $x \mapsto \#\{i : \text{dom } s \mid s(i) = x\}\}$

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



or horizontally, for the same example

$$S \hat{=} [x : N; y : \text{seq } N \mid x \leq *y] .$$

Use in signatures after $\forall, \lambda, \{ \dots \}$, etc.:

$$(\forall S \cdot y \neq \langle \rangle) \hat{=} (\forall x : N; y : \text{seq } N \mid x \leq *y \cdot y \neq \langle \rangle) .$$

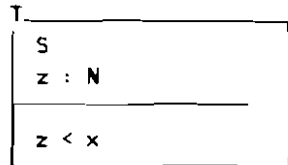
Schemas as types: when a schema name S is used as a type it stands for the set of all objects described by the schema, $\{S\}$. For example, $w : S$ declares a variable w with components x (of type N) and y (of type $\text{seq } N$) such that $x \leq *y$.

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: $w.x \leq *w.y$. Additionally, given $w : X \rightarrow S$, $w \# (\lambda S.x)$ is a function $X \rightarrow N$, etc.

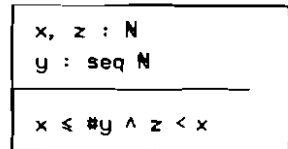
θS The tuple formed from a schema's variables: for example, θS is (x, y) . Where there is no risk of ambiguity, the θ is sometimes omitted, so that just "S" is written for " (x, y) ".

pred S The predicate part of a schema: e.g. pred S is $x \leq *y$.

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,



is



$S \mid P$ The schema S with P conjoined to its predicate part. E.g., $(S \mid x > 0)$ is $[x : N; y : \text{seq } N \mid x \leq *y \wedge x > 0]$.

$S ; D$ The schema S with the declarations D merged with the declarations of S . For example, $(S ; z : N)$ is $[x, z : N; y : \text{seq } N \mid x \leq *y]$.

$S[\text{new/old}]$ 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 $[z : N; y : \text{seq } N \mid z \leq *y]$ and $S\{y/x, x/y\}$ is $[y : N; x : \text{seq } N \mid y \leq *x]$.

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 $[x' : N; y' : \text{seq } N \mid x' \leq \#y']$.

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

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

$x : N$ $y : \text{seq } N$ $z : P \ N$
$x \leq \#y \wedge x \in z$

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

$x : N$ $y : \text{seq } N$ $z : P \ N$
$x \leq \#y \vee x \in z$

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking $\text{pred } S \Rightarrow \text{pred } T$ as the

predicate. E.g., $S \Rightarrow T$ is

$x : N$ $y : \text{seq } N$ $z : P \ N$
$x \leq \#y \Rightarrow x \in z$

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

$x : N$ $y : \text{seq } N$ $z : P \ N$
$x \leq \#y \Leftrightarrow x \in z$

$S \setminus (v_1, v_2, \dots, v_n)$

Hiding: the schema S with the variables v_1, v_2, \dots , 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 $[y : \text{seq } N \mid (\exists x : N \cdot x \leq \#y)]$. (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

$z : P \ N$
$(\exists x : N; y : \text{seq } N \cdot x \leq \#y \wedge x \in z)$

$S \uparrow (v_1, v_2, \dots, v_n)$

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

$$\begin{array}{l} x : N \\ y : \text{seq } N \\ \hline (\exists z : P \ N \cdot \\ \quad x \in \#y \wedge x \in z) \end{array}$$

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

$$\begin{array}{l} S \\ \hline x?, s, s', y! : N \\ \hline s' = s-x? \wedge y! = s \end{array}$$

pre S is

$$\begin{array}{l} x?, s : N \\ \hline (\exists s', y! : N \cdot \\ \quad s' = s-x? \wedge y! = s) \end{array}$$

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 @ T Overriding:
 $\hat{=} (S \wedge \neg \text{pre } T) \vee T.$

For example, given S above and

$$\begin{array}{l} T \\ \hline x?, s, s' : N \\ \hline s < x? \wedge s' = s \end{array}$$

S @ T is

$$\begin{array}{l} x?, s, s', y! : N \\ \hline (s' = s-x? \wedge y! = s \wedge \\ \quad \neg (\exists s' : N \cdot \\ \quad \quad s < x? \wedge s' = s)) \\ \vee (s < x? \wedge s' = s) \end{array}$$

which simplifies to

$$\begin{array}{l} x?, s, s', y! : N \\ \hline (s' = s-x? \wedge y! = s \wedge \\ \quad s \geq x?) \vee \\ (s < x? \wedge s' = s) \end{array}$$

S ; 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" (\wedge) of the resulting schemas, and hide the new variables. E.g., $S ; T$ is

$$\frac{x?, s, s', y! : N}{(\exists s_0 : N . s_0 = s-x \wedge y! = s \wedge s_0 < x? \wedge s' = s_0)}$$

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

S >> 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.

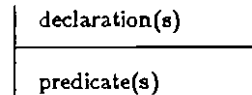
ΔS change of before to after state,
 $\equiv S$ no change of state,
 ϕS framing schema for definition of further operations.

For example

$$\begin{aligned} \Delta S &\hat{=} S \wedge S' \\ \equiv S &\hat{=} \Delta S \mid \theta S = \theta S' \\ \phi S &\hat{=} \Delta S \mid y = y' \\ S_{OP} &\hat{=} \phi S \mid x' = 0 \end{aligned}$$

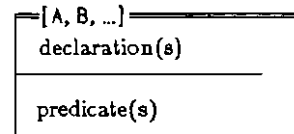
Other Definitions

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

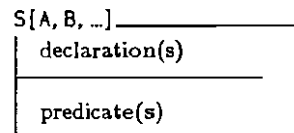


or horizontally: $D \mid P$

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



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



The following conventions are used for prefixing of schema names:

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP
8-11 Keble Road, Oxford OX1 3QD, England

Technical Monographs to August 26, 1987

- PRG-2 *Outline of a Mathematical Theory of Computation*
by Dana Scott. November 1970, 24 p., £0.50
- PRG-3 *The Lattice of Flow Diagrams*
by Dana Scott. November 1970, 57 p., £1.00
- PRG-5 *Data Types as Lattices*
by Dana Scott. September 1976, 65 p., £2.00
- PRG-6 *Toward a Mathematical Semantics for Computer Languages*
by Dana Scott and Christopher Strachey. August 1971, 43 p., £0.60
- PRG-9 *The Text of OSPub*
by Christopher Strachey and Joseph Stoy. July 1972, 2v. 126, 151 p., £3.50
- PRG-10 *The Varieties of Programming Language*
by Christopher Strachey. March 1973, 20 p., £0.50
- PRG-17 *Report on the Programming Notation 3R*
by Andrew P. Black. August 1980, 58 p., £2.30
- PRG-18 *The Specification of Abstract Mappings and their Implementation as B+ Trees*
by Elizabeth Fielding. September 1980, 74 p. + Appendix, £1.30
- PRG-20 *Partial Correctness of Communicating Processes and Protocols*
by Zhou Chao Chen and C.A.R. Hoare. May 1981, 23 p., £1.75
- PRG-22 *A Model for Communicating Sequential Processes*
by C.A.R. Hoare. June 1981, 26 p., £1.30
- PRG-23 *A Calculus of Total Correctness for Communicating Processes*
by C.A.R. Hoare. April 1981, 31 p., £1.75
- PRG-26 *The Consistency of the Calculus of Total Correctness for Communicating Sequential Processes*
by Zhou Chao Chen. February 1982, 38 p., £1.80
- PRG-29 *Specifications, Programs and Implementations*
by C.A.R. Hoare. June 1982, 29 p., £1.75
- PRG-32 *The Lispkit Manual*
by Peter Henderson, Geraint A. Jones and Simon B. Jones. 1983, 2v., 127, 136 p.,
£4.00 for both volumes
- PRG-34 *Abstract Machine Support for Purely Functional Operating Systems*
by Simon B. Jones. August 1983, 33 p. + Appendix, £1.75
- PRG-36 *The Formal Specification of a Conference Organising System*
by Tim Clemeunt. August 1983, 52 p. + Appendix, £1.75
- PRG-37 *Specification-Oriented Semantics for Communicating Processes*
by E.R. Olderog and C.A.R. Hoare. February 1984, 81 p., £1.50

- PRG-38 *Making Nets Abstract and Structured and Nets and their Relation to CSP*
by Ludwik Czaja. January/June 1984, 23, 26 p., £1.30
- PRG-42 *A Range of Operating Systems Written in a Purely Functional Style*
by Simon B. Jones. February 1985, 44 p., £1.30
- PRG-44 *The Weakest Prespecification*
by C.A.R. Hoare and He Jifeng. June 1985, 60 p., £0.85
- PRG-45 *Laws of Programming - A Tutorial Paper*
by C.A.R. Hoare, He Jifeng, I.J. Hayes, C.C. Morgan, J.W. Sanders, I.H. Sorensen, J.M. Spivey, B.A. Sufrin, A.W. Roscoe.
May 1985, 43 p., £2.35
- PRG-46 *Specification Case Studies*
by Ian Hayes. July 1985, 68 p., £2.50
- PRG-47 *Specifying the CICS Application Programmer's Interface*
by Ian Hayes. July 1985, 82 p., £3.10
- PRG-48 *CAVIAR: A Case Study in Specification*
by Bill Flinn and Ib Holm Sorensen. July 1985, 46 p., £2.00
- PRG-49 *Specification Directed Module Testing*
by Ian Hayes. July 1985, 30 p., £0.90
- PRG-50 *The Distributed Computing Software Project*
by Roger Gimson and Carroll Morgan. July 1985, 85 p., £4.00
- PRG-51 *JSD Expressed in CSP*
by K.T. Sridhar and C.A.R. Hoare. July 1985, 40 p., £1.45
- PRG-52 *Algebraic Specification and Proof of Properties of Communicating Sequential Processes*
by C.A.R. Hoare and He Jifeng. November 1985, 72 p., £0.90
- PRG-53 *The Laws of Occam Programming*
by A.W. Roscoe and C.A.R. Hoare. February 1986, 86 p., £2.50
- PRG-54 *Exploiting Parallelism in the Graphics Pipeline*
by Theoharis A. Theoharis. June 1986, 101 p., £2.50
- PRG-55 *Functional Programming with Side-Effects*
by Mark B. Josephs. Ph.D. thesis, June 1986, 101 p., £3.00
- PRG-56 *An Introduction to the Theory of Lists*
by Richard S. Bird. October 1986, 28 p., £1.50
- PRG-57 *The Pursuit of Deadlock Freedom*
by A.W. Roscoe and Naiem Dathi. November 1986, 38 p., £1.50
- PRG-58 *Formal Methods Applied to a Floating Point Number System*
by Geoff Barrett. January 1987, 47 p., £1.60
- PRG-59 Not yet allocated
- PRG-60 *The Formal Specification of a Microprocessor Instruction Set*
by Jonathan Bowen. January 1987, 72 p., £2.00