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

THE SPECIFICATION OF NETWORK SERVICES

by

Jonathan Bowen
Roger Gimson
Stig Topp-Jørgensen

Technical Monograph PRG-61

August 1987

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

The Specification of Network Services

Jonathan Bowen
Roger Gimson
Stig Topp-Jørgensen

Abstract

The specification language Z has been applied by the Distributed Computing Software Project to the formal specification of network resource managers or 'services'. The use of a formal language gives a more precise understanding of the behaviour of a service and is a prerequisite for verification of programs which use or implement the service. Additionally, the use of Z combined with informal text is sufficiently readable for the specification to be used for documentation purposes.

An introduction is provided to the style of specification devised for the project. A framework for the specification of a variety of network services has been developed. The framework is presented, and then incorporated into an example illustrating the specification of both the user's view and the implementor's view of a simple service. A discussion of the experience gained from the specification and use of the example service is also included.

Contents

1 Motivation

It is fundamental to the desigu of any complex artefact, and of computer systems in particular, that an appropriate means of describing and communicating the design is used.

A very important line of communication is between the designer and the user of the system. It is only if this communication is accomplished satisfactorily that the designer can have any expectation of meeting the requirements of the user and, likewise, the user have any expectation of being able to make proper use of the finished product.

No less important is the communication from the designer to the maker (or implementor) of the system. This is necessary to ensure that the finished product does indeed have the characteristics that the designer specified.

The aim of the work described here is the improved communication between designer, user and implementor which can be achieved by the use of formal specification in the design and documentation of computer systems.

1.1 Formal specification

Satisfactory communication relies firstly on the production of an unambiguous description. If a description is sufficiently precise, it can act as a contract between the designer, user and implementor, to ensure that they agree on what is to be provided.

A fundamental objective of the Distributed Computing Software Project has been to make use of mathematical techniques for program specification to assist the design, development and presentation of distributed system services.

The formal notation used tbroughout the project has been Z (as defined in [2-7]). This specification language, based on mathematical set theory, has been developed at the Programming Research Group over the past few years. The Distributed Computing Software Project has been testing the application of the theoretical ideas to a realistic and practical system. As a result of this, the project has been influential in the development of notational techniques which have now become a standard part of the Z style of specification.

The use of formal specification techniques, because of their rigor, tends to guide designs towards the conceptually simple. This has the advantage of making the designs easier to understand, but the possible disadvantage of making them harder to implement efficiently, since the simplest ideas do not necessarily have the most straightforward realisation.

Formal techniques encourage a level of abstraction that is important in avoiding the introduction of unnecessary implementation bias into designs. In the initial design, implementation bias simply restricts the range of possible implementations. It is usually an indication that the designer allowed unnecessary knowledge of a potential implementation to become visible at the user level.

1.2 Documentation

Conventionally, various pieces of documentation are the main means of communication between designer and user. In order that the rigor of the specifications should not be lost, it was felt to be of great importance that the system documentation should incorporate the full formalism used in the design. However, it was also important to ensure that, as for any documentation, readability and accessibility were not sacrificed in the process.

A significant amount of effort has therefore been spent on developing a manual style which combines informal and formal text. The presentation of the User Manuals emphasises the effect of each user-invoked operation on a service. The Implementor Manuals, on the other hand, concentrate on identifying the subcomponents from which an implementation of the service can be built.

2 Service specification

A service of a distributed computing system can be modelled in much the same way as a component in a centralised system.

A service can be described in terms of a service state and a set of operations which will change the state in a well-defined way. Consider a service with a state S. The effect on the service of a given operation OP can be described in terms of the preceding state S and the subsequent state S′ (the dash is used by convention in Z to denote the state after an operation). Thus, at any given time, the current state of the service can be determined from knowledge of the initial service state and of the sequence of operations executed in the lifetime of the service so far.

Two small but significant differences can exist in a distributed system, as compared to a centralised system. The first is that the individual services will usually be at least partly involved in tasks such as accounting, user authentication and access control, which would be more easily separable in a centralised system. Secondly, it is a characteristic feature of a distributed system that components in the system may continue to work after others have failed, so that the error notification and handling provided by services becomes important.

2.1 User's view

A user will in general be interested only in the externally observable behaviour of a service. In the case of a file storage service, for instance, a user will be concerned with files, filenames and file contents, but will not be interested in details of how these items are represented and stored by the service. When specifying the requirements and the user interface for a service, it is useful to do so in terms of an abstract (i.e. not implementation specific) service state and corresponding abstract operations.

If the user's view of the (abstract) service state is AS, then each abstract operation will be described in terms of the preceding and subsequent abstract states AS and AS′. In order for the state of the service to be defined at all times, the initial state of the service InitAS also needs to be established.

2.2 Implementor's view

Unlike a user, an implementor will need a much more detailed view of a service and will specifically be interested in the internal behaviour of the service. In the case of a file storage service for instance, the implementor will have to deal with items such as index blocks and data blocks.

If the implementor's view of the (concrete) service state is CS, then concrete operations are expressed in terms of the before and after states CS and CS′. As before, the initial state of the concrete service InitCS must be well-defined.

2.3 Common framework

In a distributed system consisting of a number of separate services connected by a network, it is useful for the services to have certain characteristics in common. These will include such facilities as service access, user authentication, accounting, accumulation of statistics and error reporting. Making the provision of such facilities the same across the collection of services means that the system as a whole will appear more homogeneous to the user and therefore easier to use. Also, the specification and implementation of the services becomes simpler since some parts are common to all services.

These common aspects of services have been collected together into a set of definitions known as the Common Service Framework. When required, these definitions can be incorporated into specifications of individual services in a standard way.

2.4 Correctness of implementation

In order to verify that the implementor's view of a service is compatible with the user's view of the same service, formal correctness arguments can be used.

These arguments depend on the formal definition of how the concrete and abstract representations of the service state relate to each other. In the following we will let Rel denote the relation between CS and AS, and Rel′ the same relation between CS′ and AS′.

In order for the concrete service state to be capable of representing the state of the abstract service, it needs to have at least one concrete state for each possible abstract state:

$$\forall \mathsf{AS} \cdot \exists \mathsf{CS} \cdot \mathsf{Rel}$$

And the inital concrete service state must specifically represent the the initial abstract service state:

$$\mathsf{InitCS} \Rightarrow \exists \mathsf{AS}' \cdot \mathsf{InitAS} \wedge \mathsf{Rel}'$$

For each abstract operation AOP, we must supply a corresponding concrete operation COP which is applicable in the corresponding domain to the abstract operation and which will produce a result that satisfies the abstract specification. In other words, if AOP changes the abstract state from AS to AS', then the corresponding concrete operation COP must, given an initial state CS which relates to AS according to Rel, produce a new state CS' which relates to AS' according to Rel'. This can be expressed more formally as:

$$\mathsf{pre\ AOP} \wedge \mathsf{Rel} \Rightarrow \mathsf{pre\ COP}$$
$$\mathsf{pre\ AOP} \wedge \mathsf{COP} \wedge \mathsf{Rel} \Rightarrow \exists \mathsf{AS}' \cdot \mathsf{AOP} \wedge \mathsf{Rel}'$$

The concrete state is thus considered as a data refinement of the abstract state, and each of the concrete operations must model the same behaviour on the concrete state as the corresponding abstract operation does on the abstract state.

The relationships between the two models can be illustrated as:

$$\begin{tikzcd}
\text{User view:} & \mathsf{AS} \arrow[r, "\mathsf{AOP}"] \arrow[d, leftrightarrow, "\mathsf{Rel}"'] & \mathsf{AS}' \arrow[d, leftrightarrow, "\mathsf{Rel}'"'] \\
\text{Implementor view:} & \mathsf{CS} \arrow[r, "\mathsf{COP}"] & \mathsf{CS}'
\end{tikzcd}$$

Note that operation refinement is also often applied to the concrete operations, so that each is implemented as a combination of a number of simpler operations.

3 Service documentation

In this section we give an outline of the structure we have adopted for the documentation of a service. The documentation consists of two main parts, a "User Manual" and an "Implementor Manual".

The User Manual describes the service as it appears to the user without giving unnecessary details regarding the implementation strategies used. The user manual is presented as a series of formal specifications (written in Z) interleaved with informal explanations in English prose. Apart from serving as a reference manual to the user of the finished service, this document normally also serves as the requirements specification for the service.

The purpose of the Implementor Manual is to present in detail an implementation of the service. The manual presents a concrete representation of the service state which is more directly implementable using available and (hopefully) well-documented resources, such as programming languages, file systems and databases. For each abstract operation the manual describes how the corresponding concrete operation can be refined in terms of a number of simpler operations, each of which is reasonably easy to implement in a programming language. The manual formally defines the relation between the concrete and abstract representations of the service state, which forms the basis for proofs of the correctness of each of the implemented operations. Like the User Manual, the Implementor Manual is also presented as a series of formal specifications linked together by prose which may contain additional informal explanation where required.

The manuals currently use Z throughout, and thus some effort is still required to transform the presented implementation into final code. Note that an Implementor Manual presents only one possible implementation, reflecting a particular set of design decisions. A programmer could choose to implement a service differently, provided it still satisfied the specification given in the user manual.

3.1 User manual

We have adopted the following outline for the content of user manuals:

1. Introduction — describes the purpose of the service.

2. Service state — presents the service state as observed by the user (abstract

state) together with possible constraints on change of state. Also, the initial state of the service is defined.

3. Operation parameters — defines input/output parameters which are shared by a number of service operations.

4. Reports — covers the possible reports which service operations may return, usually "Success" and a number of error cases. Each report is detailed together with the circumstances under which it will be triggered.

5. Operation definitions — describes in detail each of the operations which the service provides. The description of each operation consists of three sections:

   a. Abstract section — a possible procedure heading for the operation (as it might appear in some programming language) detailing the explicit input/output parameters, and a short informal description of the operation.

   b. Definition section — formal specification of the successful behaviour of the operation. This takes the form of a Z schema which incorporates all the formal parameters listed in the previous section. The schema may be accompanied by a short informal explanation where required. Note that the defined operation is partial and does not cater for any error conditions.

   c. Reports section — formal specification of the total operation. The total operation is formed by combining the partial operation described in the Definition section with a number of error schemas described in section 4 of the manual.

6. Service charges — presents a tariff schema defining the charges incurred by use of each of the service operations.

7. Complete service — shows how the service state and operations defined in sections 2 and 5 of the manual combine with standard states and operations defined in the Common Service Framework to form the complete service.

An example of the layout of a manual page defining a service operation is given on the next page. The formal text will be explained in more detail later.

OP

Abstract

```
OP (in?     : IN;
    out!    : OUT;
    report! : Report)
```

Informal description of the operation and the parameters.

Definition

$$
\begin{array}{|l}
\hline
\textsf{OP}_{\textsf{success}} \\
\quad \Delta S \\
\quad in? : IN \\
\quad out! : OUT \\
\hline
\quad POST\ (in?, S, out!, S') \\
\hline
\end{array}
$$

Informal text clarifying points in the formal definition of the operation.

Reports

$$
\begin{aligned}
OP \;\widehat{=}\; (OP_{success} &\wedge Success) \\
&\oplus InputError_1 \\
&\oplus InputError_2
\end{aligned}
$$

Optional informal text describing error conditions.

3.2 Implementor manual

The content of Implementor Manuals may vary considerably due to the difference of complexity in implementing various types of services. However, a typical outline for an Implementor Manual would be as follows:

1. Introduction — background and overall implementation strategy.

2. Abstract state — extract from the user manual (included to avoid cross-referencing).

3. Concrete state — defines the concrete (implementation) state together with its inital value.

4. Reports — each related to the concrete circumstances under which it will be triggered. Corresponds closely to section 4 of the user manual.

5. Operation implementations — for each of the abstract service operations, the corresponding concrete operation is defined. The description of each operation is in three sections, as in the user manual.

6. Complete service — shows how the service state and operations defined in sections 3 and 5 of the manual combine to form the complete service.

7. Implementation correctness — formally relates the abstract and concrete states as a necessary precursor to any proofs of correctness of the implementation.

Chapter 2

Common Service Framework

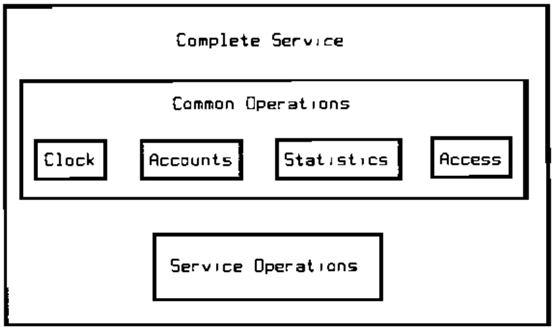
1 Introduction

When building a distributed operating system, consisting of a number of separate *services* connected to each other and to users by a communications network, there are a number of features that are common to all (or at least most) network services.

A common framework for the specification of the user interface to services is presented here, in a way which allows the common features to be factored out of the specifications of particular services.

The description is given in terms of an example skeleton service. The state and operations of this service are introduced only in outline. Any actual service description would provide explicit detail of these components. However, the example does show how any service can be elaborated to include common features.

The common features include a number of subsystems such as a service clock, accounting, statistics and access control. Each of these subsystems introduces a number of extra operations which may be performed by the service. These additional operations are introduced in separate sections which need not necessarily be absorbed in detail at a first reading, but are designed to be used for reference when required. The subsystems combine with the specific service operations to form a complete service. This is illustrated in the diagram below.



Subsequently, it is shown how a complete distributed system may be defined by combining the specification of individual services. Each service is identified by a unique user number. This allows services to act as clients to other services if required.

Finally, attributes of the network itself and the client's system are introduced, in as far as these affect the operation of a service. The 'network' may be considered to authenticate clients and can introduce errors. The client program must identify itself to the network and may also wish to keep its own accounting record.

The last section gives details of the following standard sets and data types used in service specifications. Further sets may be introduced in individual service manuals as required.

[Boolean, UserNum, UserId, Time, Interval, Money, Op, Report, Key]

1.1 Example service

A service is specified by providing a mathematical model of the state of the service, and by formally defining the change in state when an operation on the service is invoked by a client of the service.

For our example skeleton service (ES), we model the state of the service as follows.

```
ES ________________________
|   state : STATE
|--------------------
|   INV (state)
|__________________________
```

Here, STATE is a set which includes all possible states of the service. A predicate INV (the *state invariant*) is defined to hold in all valid service states.

We introduce the name ΔES to denote the change in state caused by some operation, defined as a relation between the state (undashed) before and the state (dashed) after the operation.

$$\Delta ES \;\widehat{=}\; ES \land ES'$$

Sometimes an operation leaves the state of the service unchanged.

$$\Xi ES \;\widehat{=}\; \Delta ES \mid \theta ES = \theta ES'$$

(In the following, we will assume for any schema S, unless otherwise stated, that ΔS and ΞS are defined in an equivalent way.)

The state of the service is initialised before it is ever used by any operation. The state of the skeleton service *after* initialisation is defined.

```
InitES ───────────────────────
│   ES'
├──────────────────
│   InitPRED (state')
└─────────────────────────────
```

An operation on the service is then defined by introducing the input parameters in? and output parameters out! of the operation and relating them to the change in state by further predicates. PRE is the *precondition* which must hold over the current state and the input parameters in order for the outcome of the operation to be well-defined. POST is the *postcondition* which relates the new state and output parameters to the current state and input parameters.

```
Operation_success ──────────────────────────
│   ΔES
│   in?  : IN
│   out! : OUT
├──────────────────
│   PRE  (state, in?)
│   POST (state, in?, out!, state')
└───────────────────────────────────────────
```

(Here we have specified the precondition only in terms of the initial state and inputs. In Z, this need not always be sufficient to define the domain of the operation because there may be some hidden implicit preconditions in POST. However in our style of specification, we try to avoid this for the sake of clarity.)

The sets STATE, IN, and OUT, and the predicates INV, PRE and POST used in the above definitions clearly depend on the particular operation within the particular service being specified. We leave them undefined for this example skeleton service. Additionally, the number of inputs and outputs will vary depending on the operation.

However, there are a number of attributes of operations and services, and indeed also of the network across which the service operations are invoked and of the invoking client process, which are common to all service specifications. The remainder of this document introduces these common features, and illustrates how they enable the specification of the example service to be augmented.

2 Operation attributes

As well as parameters which are particular to the service operation being specified, there are a number of other attributes common to all operations.

2.1 Reports

We add to each service operation the output parameter

```
        report! : Report
```

which indicates to the client in a standard way either that the operation succeeded or why it failed (in most cases, failure will leave the state of the service unchanged).

The normal outcome of an operation is success.

```
Success ____________________
|   report! : Report
|____________________
|
|   report! = SuccessReport
|___________________________
```

(Note that italics within formal text are used to denote implementation-specific constants.)

SuccessReport is the same across all services for simplicity. The specification of a particular operation on a particular service may introduce report values which are returned to indicate that some specific precondition of the operation has <u>not</u> been satisfied.

For example, if the specific precondition PRE of the operation on the example service is not satisfied, a report indicating the reason may be returned as follows.

```
InputError ───────────────────────
│   ΞES
│   in? : IN
│   report! : Report
├──────────────
│   ¬PRE (state, in?)
│   report! = InputErrorReport
└──────────────────────────────────
```

The state of the service is defined as being unchanged if the error occurs.

We can now define the total effect of the operation in this example as being either success, or an operation-specific error.

$$\mathrm{Operation} \mathrel{\widehat{=}} (\mathrm{Operation}_{\mathrm{success}} \land \mathrm{Success}) \lor \mathrm{InputError}$$

If more than one error report may be produced from an operation invocation, it is often useful, or even necessary, to specify an ordering of the reports according to the satisfaction of predicates over subsets of the state and inputs.

For example, if the successful outcome of an operation requires satisfaction of the predicates contained in two schemas, PRE_1 and PRE_2, then we can specify the total effect of the operation using schema overriding (assuming error schemas $\mathrm{InputError}_1$ and $\mathrm{InputError}_2$ are defined for the negation of the respective predicates, as for InputError above)

$$\begin{aligned}\mathrm{Operation} \mathrel{\widehat{=}} (\mathrm{Operation}_{\mathrm{success}} &\land \mathrm{Success})\\ &\oplus \mathrm{InputError}_1\\ &\oplus \mathrm{InputError}_2\end{aligned}$$

which can be expanded to give the following.

$$\begin{aligned}\mathrm{Operation} \mathrel{\widehat{=}} ((((\mathrm{Operation}_{\mathrm{success}} &\land \mathrm{Success})\\ &\land \mathrm{PRE}_1) \lor \mathrm{InputError}_1)\\ &\land \mathrm{PRE}_2) \lor \mathrm{InputError}_2\end{aligned}$$

In other words, from the report produced by calling this operation we could deduce:

$$\begin{array}{lcr}\mathrm{Operation} \mid \mathrm{report!} = \mathit{Success} & \Rightarrow & \mathrm{PRE}_1 \land \mathrm{PRE}_2\\ \mathrm{Operation} \mid \mathrm{report!} = \mathit{InputErrorReport}_1 & \Rightarrow & \neg\mathrm{PRE}_1 \land \mathrm{PRE}_2\\ \mathrm{Operation} \mid \mathrm{report!} = \mathit{InputErrorReport}_2 & \Rightarrow & \neg\mathrm{PRE}_2\end{array}$$

Note that if overriding is used to specify the total operation, then it is not necessart to explicitly include the preconditions in the definition of the successful operation.

At this point it is worth noting that, in practice, a client will usually invoke a particular operation on a service by calling a programming language construct, such as a procedure. The procedure takes input parameters to be passed to the service and output parameters returned by the service as a result of the call. The output parameters will include the report value.

We include in the user manual for the service an indication of the format of the procedure call that would be used in a procedure-oriented interface as follows.

```
Operation (in?     : IN;
           out!    : OUT;
           report! : Report)
```

2.2 Client identification

In order that a service can attribute any resources used in performing an operation to a particular user, each client is given a *user number* which is allocated from the set UserNum. The allocation is public – that is, it is common for clients to know each others' user numbers. It is expected that the user number of a particular client will change only rarely, if at all.

The user number of the client who invoked an operation is assumed to be an implicit parameter of the operation. In other words, the user number is not explicitly passed as a parameter to the service, but is derived from other information (see section 4.1 on Authentication).

We therefore augment the attributes of an operation with the user number of the client.

```
clientnum : UserNum
```

2.3 Special clients

Some service operations may behave differently if their invoking client is either one of two special cases.

The *guest user* is an identity which may be assumed by any client who is (usually only temporarily) lacking their own individual identity. The guest user has the fixed user number *GuestNum* which is a special value from the set of all user numbers.

The *service manager* is a particular user, fixed for any particular service, who is responsible for the management of the resources provided by the service.

$$\{GuestNum,\ ManagerNum\} \subset \mathsf{UserNum}$$

Some service operations which need to attribute the use of resources to an identifiable client may prevent the guest user from successfully performing the operation. The following schema may be used as an overriding component in an operation definition to ensure that the client is known, or to produce an appropriate error report otherwise.

```
NotKnownUser ____________________________
|   clientnum : UserNum
|   report! : Report
|____________________
|   clientnum = GuestNum
|   report! = NotKnownUserReport
|________________________________________
```

Such an operation would be specified to leave the state of the service unchanged if the user is not properly identified.

$$\mathsf{Operation} \mathrel{\widehat{=}} \mathsf{Operation}_{\mathsf{success}} \oplus (\mathsf{NotKnownUser} \land \Xi \mathsf{ES})$$

Similarly, services often include special operations which are invoked to manage the resources provided by the service. Examples of such operations include status operations to discover the amount of a resource currently being used by each client, or scavenge operations to reclaim resources that are no longer being used. The service may restrict successful invocation of these operations to the service manager by using the following overriding schema.

```
NotManager ______________________________
|   clientnum : UserNum
|   report! : Report
|____________________
|   clientnum ≠ ManagerNum
|   report! = NotManagerReport
|________________________________________
```

A management operation would therefore be specified as follows.

$$\mathsf{Operation} \mathrel{\widehat{=}} \mathsf{Operation}_{\mathsf{success}} \oplus (\mathsf{NotManager} \wedge \Xi\mathsf{ES})$$

2.4 Current time

Each service has access to the current time (for example via access to a common time service). It is useful to denote this by including as an implicit attribute of an operation the time now at which the operation was invoked.

$$\mathsf{now} : \mathsf{Time}$$

We do not attempt here to specify in more detail the value that this attribute will assume, except to informally hint that for successive operations the value will be non-decreasing! Later, a standard specification for a service clock is presented, which may be used in individual services if desired.

2.5 Operation cost

Each service is responsible for charging its clients for their use of the resources provided by the service. Every operation has an output parameter which indicates the cost incurred by the client in performing the operation. (We shall see later that this parameter need not be explicitly included in the procedure call when using a procedure-oriented interface.)

$$\mathsf{cost!} : \mathsf{Money}$$

The value of this parameter will be specified separately (see section 3.1 on Service charges).

2.6 Key-linked operations

Some operations are designed to operate over a potentially large set of values (such as block identifiers). Such operations are designed to allow the set 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.

For example, say an Operation schema requires the traversal of the (potentially large) set

$$xs : \mathbb{F}\ X$$

(For example, X could be the set of all file identifiers.)

The operation itself is designed to traverse only a subset of xs on each call, 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 Key. Each 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*:

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

The client then continues to call Op repeatedly, supplying as the new key in each case the key returned by the previous call. The following is an example of the i^{th} call:

$$\begin{aligned} Operation \mid\ & key? = key_i \quad \wedge \\ & key! = key_{i+1} \quad \wedge \\ & subxs = subxs_i \end{aligned}$$

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

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

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

$$xs = \bigcup_i subxs_i$$

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

```
=[X]═══════════════════════════
   KeySet : Key ⇸ 𝔽 X
├──────────────
   KeySet(StartKey) = ∅
   KeySet(EndKey)   = X
└──────────────────────────────
```

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 xs).

The following framing schema (parameterised by an appropriate set X) is used to simplify the definition of such key-linked operations.

```
ΦKey[X]─────────────────────────────────
   key?   : Key
   key!   : Key
   xs     : 𝔽 X
   subxs  : 𝔽 X
├──────────────
   KeySet(key?) ⊂ KeySet(key!)
   subxs = (KeySet(key!) \ KeySet(key?)) ∩ xs
└───────────────────────────────────────
```

[Note: a *framing schema* is denoted with the prefix letter 'Φ' and consists of a partial specification for use as part of a subsequent schema.]

The difference between the sets denoted by the two keys indicates the subset of xs involved in the particular call.

BadKey indicates that an input key has been provided which does not denote a valid set. Note that this includes supplying an end key to an operation. This error schema should always be provided by key-linked operations.

```
BadKey ────────────────────────────
   key?    : Key
   report! : Report
├──────────────
   key? ∉ (dom KeySet) \ EndKey
   report! = BadKeyReport
└──────────────────────────────────
```

2.7 Operation identification

So far, we have only considered the specification of individual operations on the example service. It is useful to define the effect of any general operation on a service.

In order to select which operation is to be performed, the operations of a service are each identified by a different value from the set Op, and each call includes an explicit parameter denoting the operation to be invoked. (When using a procedure-oriented interface, this parameter is implied by the name of the procedure being called.) We can define a framing schema for an operation.

```
ΦOp ____________
| op? : Op
|_______________
```

If the individual operations on the example service have been specified as A, B, ..., D, with operation numbers *AOp*, *BOp*, ..., *DOp* respectively, then the effect of an arbitrary choice of one of these operations on the service can be specified as follows.

```
ESServiceOps ≙
    (A    ∧ ΦOp | op? = AOp    ) ∨
    (B    ∧ ΦOp | op? = BOp    ) ∨
          ...
    (D    ∧ ΦOp | op? = DOp    )
```

Any attempt to invoke a non-existent operation on a particular service is rejected with an appropriate report, which will be included in a later definition of the complete service.

```
BadOperation ______________________
| report! : Report
|__________________
| report! = BadOperationReport
|__________________________________
```

2.8 Operation parameters

Typically, for each operation requested by clients there is an output parameter reporting the outcome of the operation (report!). Additionally the current time (now) and the user number of the client (clientnum) are available. It is convenient to define a schema containing such parameters in each service user manual.

```
φBasicParams ________________________
|   report!   : Report
|   now       : Time
|   clientnum : UserNum
|____________________________________
```

Additionally, the basic parameters are supplemented by *hidden* parameters, normally an operation identifier and the cost of executing the operation.

```
φParams ______________________
|   φBasicParams
|   op?    : Op
|   cost!  : Money
|_____________________________
```

Again, it may be convenient to define such a schema in a user manual. Note that these hidden parameters will not normally appear as parameters to procedures invoked in a specific user programming language to execute the operation, but will be passed by some other means. For example, op? will depend on the *name* of the procedure called by the user.

Note that since φParams includes op?, it may be used instead of φOp in the definition of ESServiceOps if this is convenient in a particular service.

3 Service attributes

Having specified the effect of individual operations on a service, it is then possible to consider the attributes that apply to the service as a whole. These include charges, a null operation, and four subsystems, each with their own state and operations, which may be incorporated into the specifications of individual services.

3.1 Service charges

Each service operation will incur some charge on the invoking client. The charge may be fixed or may be a function of the parameters of the operation. (Some service operations may sometimes give a credit because of resources returned by the client; this is indicated by a negative charge.)

A service manual will include a *tariff* section which defines the value of the cost! parameter for any particular invocation of an operation. The details of the tariff will be specific to a particular implementation of a service.

For example, a tariff of the following form may be imposed on a client of the example service who successfully invokes an operation.

$$
\begin{array}{|l}
\textsf{ESTariff} \\
\hline
\quad \Delta ES \\
\quad op? \;:\; Op \\
\quad in?_A \;:\; IN_A \\
\quad out!_A \;:\; OUT_A \\
\quad \ldots \\
\quad cost! \;:\; Money \\
\hline
\quad op? = AOp \Rightarrow cost! = ABasicCost + \\
\qquad\qquad\qquad\qquad AExtraCost(\Delta ES, in?_A, out!_A) \\
\quad op? = BOp \Rightarrow cost! = BBasicCost + \ldots \\
\quad \ldots \\
\hline
\end{array}
$$

where $\{\, ABasicCost, BBasicCost, \ldots \} \subseteq Money$
$AExtraCost \in (\Delta ES \times IN_A \times OUT_A) \twoheadrightarrow Money$
etc.

The cost when errors occur should also be covered and included in the definition of a tariff error schema.

```
ErrorTariff ________________
|  cost! : Money
|--------------------
|  cost! = ErrorCost
|___________________________
```

```
ΦESTariff  ≙  Success ⇒ ESTariff    ∧
             ¬Success ⇒ ErrorTariff
```

Note that a particular service may specify a more complex set of charges for different error reports. This tariff framing schema combines with the service operations to define the basic service:

```
ESBasicOps  ≙  ΦESTariff ∧ ESServiceOps
```

3.2 Null operation

A null operation is provided in most services. This operation does not change the state of the service, but allows any client to check that they can successfully access the service. A standard (small) cost is involved.

```
Null ______________________
|  ΦParams
|  Success
|--------------------
|  op? = NullOp
|  cost! = NullCost
|__________________________
```

Note that at this stage we do not know the state of the particular service. Hence the fact that the service state does not change will be recorded when the complete service is formally defined.

3.3 Service clock subsystem

A service may include its own clock subsystem which maintains the current time.

```
Clock ______________
|   now : Time
|______________
```

Initially the clock is set to some value (typically using a separate Time Service, although this is not specified here).

```
InitClock ______________
|   inittime : Time
|   Clock'
|______________
|   now' = inittime
|______________
```

The interval between most service operations as measured by the clock is positive, but may be zero if the granularity of time measurement is large.

```
ΦClock ______________
|   ΔClock
|______________
|   now' ≥ now
|______________
```

Note that now is considered to be the time the service operation took place, and now' will be the time the next operation will take place. Thus now' will not be available in the specifications of operations in practice.

Two operations are associated with this subsystem. The current time according to the service can be read by any client. A (small) fixed cost is associated with this operation.

```
GetClock_success ______________
|   ΦClock
|   now!  : Time
|   cost! : Money
|______________
|   now! = now
|   cost! = GetClockCost
|______________
```

```
GetClock  ≙  GetClock_success ∧ Success
```

When the time is set, the time of the next operation will then be after (or possibly the same as) the required time.

```
SetClock_success ──────────────────┐
│   ΔClock
│   now?  : Time
│   cost! : Money
├──────────────────
│   now' ≥ now?
│   cost! = SetClockCost
└──────────────────────────────────┘
```

```
SetClock  ≙  (SetClock_success ∧ Success)
                  ⊕ (NotManager ∧ ΦClock ∧ ErrorTariff)
```

The clock may only be set by the service manager, but may be read by any client. These clock subsystem operations may be combined as follows:

```
ClockOps  ≙
    (GetClock ∧ ΦOp | op? = GetTimeOp ) ∨
    (SetClock ∧ ΦOp | op? = SetTimeOp )
```

3.4 Service accounting subsystem

Each service may keep an accounting record of the accumulated credit and charges made to each client for use of that service. Account balances may be positive denoting a credit or negative denoting a debt. Here we assume that the service can keep a record for all possible users, so the accounts function is total. (If the number of possible users were very large, this may not be feasible in practice.)

```
Accts ────────────────────────────────┐
│   accounts : UserNum → Money
└─────────────────────────────────────┘
```

Initially the accounts are all zero.

```
InitAccts  ≙  Accts' | ran accounts' = {0}
```

When a service operation is performed, the output cost parameter (cost!) is deducted from the balance for the appropriate client. The following framing schema will therefore be included in the specification of each service operation.

```
ΦAccts ──────────────────────────────────────────
│ ΔAccts
│ cost!      : Money
│ clientnum : UserNum
├──────────────
│ accounts' = accounts ⊕
│   {clientnum ↦ accounts(clientnum) − cost!}
└─────────────────────────────────────────────────
```

A subsystem operation is provided to allow clients to check the balance of their account. Note that this operation involves a cost itself (similar to a charge for a bank statement). The balance is that after deduction of this amount.

```
GetBalance_success ──────────────────────
│ ΦAccts
│ balance!   : Money
│ clientnum : UserNum
├──────────────
│ balance! = accounts'(clientnum)
│ cost! = GetBalanceCost
└────────────────────────────────────────
```

```
GetBalance  ≙  (GetBalanca_success ∧ Success)
```

A management operation is provided to check the accounts of those clients with non-zero balances.

```
CheckAccounts_success ──────────────────
│ ΞAccts
│ accounts! : UserNum ⇸ Money
├──────────────
│ accounts! = accounts ⩥ {0}
└───────────────────────────────────────
```

```
CheckAccounts  ≙  (CheckAccounts_success ∧ Success)
                       ⊕ (NotManager ∧ ΦAccts ∧ ErrorTariff)
```

The manager may reinitialise all the accounts if required.

```
ZeroAccounts_success ──────────────────
│ ΔAccts
├──────────────
│ ran accounts′ = {0}
└──────────────────────────────────────
```

```
ZeroAccounts  ≙  (ZeroAccounts_success ∧ Success)
                    ⊕ (NotManager ∧ ΦAccts ∧ ErrorTariff)
```

Finally, a specified account may be credited when a client pays all or part of his bill, or prepays for use of the service.

```
CreditAccount_success ─────────────────────────────────
│ ΔAccts
│ clientnum? : UserNum
│ credit?    : Money
├──────────────
│ accounts′ = accounts ⊕
│   {clientnum? ↦ accounts(clientnum?) + credit?}
└──────────────────────────────────────────────────────
```

```
CreditAccount  ≙  (CreditAccount_success ∧ Success)
                    ⊕ (NotManager ∧ ΦAccts ∧ ErrorTariff)
```

Apart from GetBalance, these operations can only be invoked by the service manager and no cost is involved, unless a client who is not the service manager attempts the operation in which case the error charge will be incurred.

The operations combine to form the accounting subsystem operations:

```
AcctsOps  ≙
    (GetBalance     ∧ ΦOp | op? = GetBalanceOp     ) ∨
    (CheckAccounts  ∧ ΦOp | op? = CheckAccountsOp  ) ∨
    (ZeroAccounts   ∧ ΦOp | op? = ZeroAccountsOp   ) ∨
    (CreditAccount  ∧ ΦOp | op? = CreditAccountsOp )
```

This subsystem could be augmented to impose a credit limit if desired, but this would require an extra error report.

3.5 Service statistics subsystem

Each service may keep a record of the number of invocations of each of its operations.

```
Stats ─────────────────────────────
│   calls : Op → OpCount
└──────────────────────────────────
```

Initially, the number of calls for all operations is zero.

```
InitStats ≙ Stats′ | ran calls′ = {0}
```

When a service operation is performed, the call count for that operation is incremented. The following framing schema will therefore be included in the specification of each service operation.

```
φStats ──────────────────────────────────────────
│   ΔStats
│   op? : Op
├────────────────
│   calls′ = calls ⊕ {op? ↦ (calls op?) + 1}
└────────────────────────────────────────────────
```

Management operations are provided on the subsystem to check the non-zero counts and to zero the accumulated statistics.

```
CheckStats_success ─────────────────
│   ΞStats
│   calls! : Op ⇸ OpCount
├────────────────
│   calls! = calls ⩥ {0}
└───────────────────────────────────
```

```
CheckStats ≙ (CheckStats_success ∧ Success)
                     ⊕ (NotManager ∧ ΞStats)
```

```
ZeroStats_success ──────────────────
│   ΔStats
├────────────────
│   ran calls′ = {0}
└───────────────────────────────────
```

```
ZeroStats  ≙  (ZeroStats_success ∧ Success)
                      ⊕ (NotManager ∧ ≡Stats)
```

These statistics subsystem operations can only be invoked by the service manager.

```
StatsOps  ≙
    (CheckStats ∧ ΦOp | op? = CheckStatsOp ) ∨
    (ZeroStats  ∧ ΦOp | op? = ZeroStatsOp  )
```

3.6 Service access subsystem

For some sequences of management operations it is important to ensure that the state of the service is not changed, or even observed, by other clients between operations. It is therefore possible in some services for the service manager to enable or disable access to the service by other clients.

The state of the access subsystem includes an indication of whether service access to other clients is enabled or not. Initially service access is not enabled.

```
Access ________________________
|   enabled : Boolean
|______________________________
```

```
InitAccess  ≙  Access' | enabled' = False
```

Operations on the basic service can only be performed if it is enabled or if the operations are performed by the service manager.

```
ΦAccess _____________________________
|   ≡Access
|   clientnum : UserNum
|___________________
|   enabled = True ∨
|   clientnum = ManagerNum
|____________________________________
```

Operations on the basic service will fail with an error report if access is disabled and it is not the service manager performing them.

```
NotEnabled ──────────────────────────────┐
│   ΞAccess
│   clientnum : UserNum
├──────────────────────
│   enabled = False
│   clientnum ≠ ManagerNum
│   report! = NotEnabledReport
└────────────────────────────────────────┘
```

Management operations are provided on the subsystem to change the state of access.

```
Enable_success ───────────┐
│   ΔAccess
├──────────────────
│   enabled' = True
└─────────────────────────┘
```

```
Enable  ≙  (Enable_success ∧ Success)
               ⊕ (NotManager ∧ ΞAccess)
```

```
Disable_success ──────────┐
│   ΔAccess
├──────────────────
│   enabled' = False
└─────────────────────────┘
```

```
Disable  ≙  (Disable_success ∧ Success)
               ⊕ (NotManager ∧ ΞAccess)
```

These operations specific to changing the state of access can only be invoked by the service manager.

```
AccessOps  ≙
    (Enable  ∧ ΦOp | op? = EnableOp   ) ∨
    (Disable ∧ ΦOp | op? = DisableOp  )
```

3.7 Service operations

This completes the definitions of the common operations that may be available on a service. Not every service need implement the subsystems for a local clock, accounting, statistics or access control. Every service must implement the null operation.

An operation may occasionally fail, even if its preconditions are satisfied, because of an underlying nondeterministic fault in its implementation (for example, a hardware fault or the unavailability of some other service). In this case a standard failure report is returned.

```
ServiceError ____________________
|   fault   : Boolean
|   report! : Report
|_______________
|
|   fault = True
|   report! = ServiceErrorReport
|________________________________
```

The state of the service should remain unchanged in this case.

Note that this imposes a heavy, if not impossible, burden on the implementor of the service to ensure recovery from all such errors without changing the observable service state. An alternative, but not very useful, specification would allow the service to assume any valid state after such an error. In the case of a catastrophic error such as complete disk failure, the implementation could be designed to continually return ServiceError and so not have to return the previous state!

We are now in a position to specify all the operations and error conditions for our example service. In this example we shall define a service including a null operation, a clock, accounting, statistics and access control.

The combined state of the complete service is:

```
ESState  ≙  ES ∧ Clock ∧ Accts ∧ Stats ∧ Access
```

The initial state is defined as:

```
InitESState  ≙
    InitES ∧ InitClock ∧ InitAccts ∧ InitStats ∧ InitAccess
```

The possible changes of state of the complete service, covering all operations which can be performed by the service, including the case where the service is not enabled, is as follows:

```
ESAllOps  ≙
  (ESBasicOps  ∧ ΔES ∧ ΦClock ∧ ΦAccts ∧ ΦStats ∧ ΦAccess) ∨
  (Null        ∧ ≡ES ∧ ΦClock ∧ ΦAccts ∧ ΦStats ∧ ΦAccess) ∨
  (ClockOps    ∧ ≡ES ∧ ΔClock ∧ ΦAccts ∧ ΦStats ∧ ΦAccess) ∨
  (AcctsOps    ∧ ≡ES ∧ ΦClock ∧ ΔAccts ∧ ΦStats ∧ ΦAccess) ∨
  (StatsOps    ∧ ≡ES ∧ ΦClock ∧ ≡Accts ∧ ΔStats ∧ ΦAccess) ∨
  (AccessOps   ∧ ≡ES ∧ ΦClock ∧ ≡Accts ∧ ≡Stats ∧ ΔAccess) ∨
  (NotEnabled  ∧ ≡ES ∧ ΦClock ∧ ≡Accts ∧ ≡Stats ∧ ≡Access)
```

Finally, we include the possibility of a bad operation number (the only possible conclusion if all the operation preconditions have failed) or a non-deterministic service error:

```
ESOps  ≙
 ((BadOperation ∧ ≡ES ∧ ΦClock ∧ ≡Accts ∧ ≡Stats  ∧ ≡Access) ⊕
  (ESAllOps     ∧ ΔES ∧ ΔClock ∧ ΔAccts ∧ ΔStats  ∧ ΔAccess))
 ∨
  (ServiceError ∧ ≡ES ∧ ΦClock ∧ ≡Accts ∧ ≡Stats  ∧ ≡Access)
```

Similar schemas should be defined at the end of each service manual.

3.8 Service identification

So far, we have only considered the specification of an individual service. We have made use of a number of definitions which are specific to the service in question, such as ES, ESTariff and ESBasicOps.

It is useful to define the effect of any general operation on the complete collection of services. Since services may act as clients by invoking operations on other services, they are given user numbers from the same set UserNum as other users.

In order to select which service is to be affected by a particular operation, the user number of the service is provided as a parameter. (When using a procedure-oriented interface, the service to be affected is implied by the name of the procedure being called.)

```
ΦSv______________________
|    sv? : UserNum
|________________________
```

Any attempt to invoke an operation on a non-existent service is rejected with an appropriate report.

```
BadService ──────────────────────┐
│   report!  : Report
├──────────────────
│   report! = BadServiceReport
└────────────────────────────────┘
```

If the individual service states have been specified as WSState, XSState, ..., ZSState, the initial states are WSInitState, XSInitState, ..., ZSInitState, and the combined operations on the services are defined as WSOps, XSOps, ..., ZSOps, then the combined state, the initial state and the effect of an arbitrary operation on an arbitrary service on the network can be specified as follows:

```
SvState  ≙
    WSState ∧ XSState ∧ ... ∧ ZSState

SvInitState  ≙
    WSInitState ∧ XSInitState ∧ ... ∧ ZSInitState

SvOps  ≙
    (BadService ∧ ΞSvState ) ⊕
    ((WSOps ∧ ΞSvState\WSState ∧ ΦSv | sv? = WSv ) ∨
     (XSOps ∧ ΞSvState\XSState ∧ ΦSv | sv? = XSv ) ∨
          ...
     (ZSOps ∧ ΞSvState\ZSState ∧ ΦSv | sv? = ZSv ))
```

In other words, any operation on a particular service does not affect the state of any other service. Note that in the case of services making use of common operations, the common state components should be renamed uniquely for each service to avoid name clashes.

(Strictly speaking, the invocation of a service operation may cause operations on <u>other</u> services to be performed by the invoked service. However, as far as the client of the original operation is concerned, those additional operations can be considered as having been performed by other users after the completion of the original operation and before the client can perform another operation on any affected service. Hence, because no user has control over the interleaving of other users' operations between their own, there is no need to explicitly cater for these secondary effects in the specification.)

4 Network attributes

There are some features of services that are independent of the particular service upon which an operation is being performed. We include authentication of clients at this stage, since it is something which can be considered the responsibility of the network, rather than of an individual service (otherwise, for example, a service could impersonate one of its clients). Indeed, it is possible for networks to include security and authentication measures as part of the hardware network interface.

4.1 Authentication

Authentication ensures that the client of a service operation is genuine, so that any costs incurred in performing the operation can be reliably attributed to a particular client. A very simple scheme has been chosen which makes it difficult for one client to impersonate another.

So far, we have presumed that the user number of the client is an implicit parameter of any service operation. Since user numbers are public, they do not provide a secure identification of the client.

In order to provide authentication, each registered client also has a *user identifier*. User identifiers are allocated privately, from the set UserId; a client should not reveal his user identifier to anyone else. Since user identifiers may become compromised (known by too many people) or forgotten (known by too few!), it might be necessary to change a client's user identifier from time to time.

Authentication is achieved by the existence of a (secret) partial function

$$\begin{array}{|l}
\mathsf{authentic} : \mathsf{UserId} \nrightarrow \mathsf{UserNum} \\
\hline
(\mathit{GuestId} \mapsto \mathit{GuestNum}) \in \mathsf{authentic}
\end{array}$$

which for any user identifier gives the user number of the client who should be its sole possessor. Since the set UserId of user identifiers has been made very large, and the set (dom authentic) of *authentic* user identifiers has been made a relatively small part of it, it will be hard for clients to guess the user identifiers of others.

We have already introduced the guest user, which some services might recognise as a special client, and who has the user number *GuestNum*. The guest user has the user identifier *GuestId*. This user identifier is public, and is expected to be used by clients temporarily without a private user identifier of their own. The guest user is always authentic.

Each service operation has an explicit input parameter

```
clientid? : UserId
```

identifying the client who has invoked the operation. (We will see later that in a procedure-oriented interface this parameter need not be provided explicitly by the client on each call.)

The authentication performed by the service-network interface will reject an operation if the client is not authentic. If authentication is successful, the user number of the client (clientnum) is defined and may be used in specifying the particular behaviour of the operation, as already described.

```
IsAuthentic ________________________________
|   clientid? : UserId
|   clientnum : UserNum
|___________________
|   clientnum = authentic clientid?
|___________________________________________
```

```
NotAuthentic _______________________________
|   clientid? : UserId
|   report!   : Report
|___________________
|   clientid? ∉ dom authentic
|   report! = NotAuthenticReport
|___________________________________________
```

We augment the specification of the service operations as follows.

```
SvAuthOps ≙
    (IsAuthentic ∧ SvOps) ∨
    (NotAuthentic ∧ ΞSvState)
```

4.2 Network errors

In rare cases an unexpected network failure may occur during the transmission of parameters to or from a service operation. In this case, an error report is returned, but the client cannot determine from the report whether or not the operation was executed.

```
NetError ________________________________
| report! : Report
|______________
|
| report! = NetErrorReport
|________________________________________
```

Again the service operations are augmented. (We use the notation S\(*!) to denote the schema S with all output parameters, ending in !, hidden.)

```
NetOps ≙
    SvAuthOps ∨
  ((SvAuthOps\(*!) ∨ ΞSvState) ∧ NetError)
```

Hence no meaning can be attributed to any output parameters if a network error has occurred, except the error report! value itself. However we guarantee that the operation either will or will not have taken place (sometimes known as 'at-most-once semantics').

5 Client attributes

So far, we have defined the following explicit parameters as common to each operation invocation.

```
Params ______________________________
| clientid? : UserId
| sv?       : UserNum
| op?       : Op
| cost!     : Money
| report!   : Report
|_____________________________________
```

These parameters, plus the input and output parameters specific to an operation, must all be present if we view the interface to a service at a low enough level (for example as data transmitted over a network).

However, in a procedure-oriented interface we have already said that the identification of the service sv? and the operation op? is implied by the name of the procedure itself.

In order to reduce the number of parameters that must be explicitly provided on each procedure call still further, it is convenient to specify that the client's name and the accumulated cost incurred are stored locally in the client program.

5.1 Client identification

Since the identification of a client program (or process, or operating system environment) is likely to remain constant over a number of service calls, it is convenient to allow the user identifier of the current client to be remembered in the state of the client program.

```
Client ___________________________
| clientid : UserId
|__________________________________
```

Initially, on starting a new client program, the current client is the guest user.

InitClient $\widehat{=}$ Client′ | clientid′ = *GuestId*

The current client may he changed or interrogated by operations. (Note that these operations are local to the client program, rather than being performed by a service, and so do not involve the normal service parameters. Their effect is left non-deterministic in this specification since their use is completely under the control of the client.)

```
SetClientId ____________________________
|   ΔClient
|   newclientid? : UserId
|_______________
|
|   clientid' = newclientid?
|_______________________________________
```

```
GetClientId ____________________________
|   ≡Client
|   currentclientid! : UserId
|_______________
|
|   currentclientid! = clientid
|_______________________________________
```

```
LocalClientOps  ≙  SetClientId ∨ GetClientId
```

When calling a service operation, the input parameter clientid? is that of the current name remembered by the client program. The following framing schema will therefore be used in the specification of each service operation.

```
ΦClient ______________________
|   ≡Client
|   clientid? : UserId
|_______________
|
|   clientid? = clientid
|_____________________________
```

5.2 Client accounting

It is also convenient to accumulate the cost incurred by the client's use of a service in the client program, rather than pass it explicitly as an output parameter on each procedure call. We therefore allow the client program to accumulate the total costs incurred over a number of service operations.

```
Cost ________________________
|   totalcost : Money
|____________________________
```

Initially, on starting a new client program, the accumulated cost is zero.

```
InitCost  ≙  Cost′ | totalcost′ = 0
```

The accumulated cost may be interrogated or reset to zero by operations. (Note that these operations are local to the client program, rather than being performed by a service, and so do not involve the normal service parameters.)

```
GetCost ______________________
|   ΞCost
|   totalcost! : Money
|___________________
|   totalcost! = totalcost
|_____________________________
```

```
ZeroCost _____________________
|   ΔCost
|___________________
|   totalcost′ = 0
|_____________________________
```

```
LocalCostOps  ≙  GetCost ∨ ZeroCost
```

When calling a service operation, the output cost parameter (cost!) is added to the accumulated cost remembered by the client program. Some reports denote that access to a service has not been possible, so no cost has been incurred, and the cost! parameter is undefined.

```
NoCostReports  ≙  {ServiceErrorReport,
                    BadServiceReport,
                    NotAuthenticReport,
                    NetErrorReport}
```

The following framing schema will then be used in the specification of each service operation.

```
ΦCost ______________________________________
|   ΔCost
|   cost! : Money
|   report! : Report
|___________________
|   (report! ∉ NoCostReports) ⇒
|       totalcost' = totalcost + cost!
|   (report! ∈ NoCostReports) ⇒
|       totalcost' = totalcost
|____________________________________________
```

It is possible, if a network error has occurred and if the invoked operation was indeed performed by the service, that some actual charge may have been incurred by the client. Clients can check their actual charges by invoking the `GetBalance` operation on the appropriate service.

5.3 Client view

The client views the state of the whole system as including the local authentication and accounting operations. Note that initially, the state of the network services may be any valid state.

```
LocalState  ≙  SvState ∧ Client ∧ Cost

InitLocalState  ≙  SvState ∧ InitClient ∧ InitCost

LocalOps  ≙
    (NetOps           ∧ ΔSvState ∧ ΦClient ∧ ΦCost) ∨
    (LocalClientOps   ∧ ΞSvState ∧ ΔClient ∧ ΞCost) ∨
    (LocalCostOps     ∧ ΞSvState ∧ ΞClient ∧ ΔCost)
```

6 Sets and data types

Service specifications, and the common service framework presented here, make use of a number of given sets and a data type. These are described in this section.

6.1 Boolean values

A boolean data type is sometimes useful when there is a simple yes/no choice, defined as follows.

$$\mathsf{Boolean} ::= \mathit{False} \mid \mathit{True}$$

6.2 User numbers and user ids

The set UserNum is a finite set of publicly known "numbers" associated with clients. The set UserId is a corresponding finite set of private identifiers for clients. (Note that there *may* be more than one valid UserId associated with a given UserNum. Also each service has a UserNum, and so may act as a client to another service.)

6.3 Time and intervals

The set Time denotes the finite set of all instants of time (to an appropriately small resolution, such as a second) covering dates relevant to the life of the system.

The set Interval denotes a finite set of non-negative time intervals, or differences between pairs of time instants.

The follow infix operators and constants are assumed to be defined for Time and Interval:

$$\begin{array}{|l}
\_ \geq \_ \ : \mathsf{Time} \leftrightarrow \mathsf{Time} \\
(\_ + \_) : (\mathsf{Time} \times \mathsf{Interval}) \rightarrow \mathsf{Time} \\
(\_ - \_) : (\mathsf{Time} \times \mathsf{Interval}) \rightarrow \mathsf{Time} \\
\mathit{ZeroTime} : \mathsf{Time} \\
\hline
\_\geq\_ \in \mathsf{total\_order}\ \mathsf{Time} \\
\mathit{ZeroTime} = \mathsf{min}\ \mathsf{Time}
\end{array}$$

```
│    _ ≥ _   : Interval ↔ Interval
│  (_ - _) : (Time × Time) → Interval
│  ZeroInterval : Interval
├──────────────────
│  _≥_ ∈ total_order Interval
│  ZeroInterval = min Interval
│  ∀ t:Time •
│    t + ZeroInterval = t
│    t - ZeroInterval = t
│    t - t = ZeroInterval
```

All these operators are defined to be total, ignoring any problems with error conditions caused by arithmetic overflow or underflow. Note that the addition of two absolute Time instants would be meaningless.

6.4 Money

The set Money denotes a finite set of all (signed) measures of cost. It is used for operation charges and accounting purposes. The following infix operators are assumed to be defined for Money:

```
│    _ ≥ _   : Money ↔ Money
│  (_ + _) : (Money × Money) → Money
│  (_ - _) : (Money × Money) → Money
├──────────────────
│  _≥_ ∈ total_order Money
```

The operators above are defined to be total, again to avoid errors.

6.5 Operation identifiers

The set Op denotes the finite set of possible operation identifiers. These are unique for different operations within a given service. However, they may be shared across services since the user number of the service itself may be used to identify on which service a particular operation is to be performed. Common operations, such as the null operation, will be given a standard operation identifier across all services to avoid confusion.

6.6 Reports

The set `Report` denotes the finite set of possible reports which may be returned by operations. As for operation identifiers, reports need only be unique within a particular service. Again, common reports will be given standard values across all services.

6.7 Keys

The finite set `Key` is used for certain operations which are called in a sequence, passing a key value between successive operations. There are special first and last keys (*StartKey* and *EndKey*) for initialisation and termination of the sequence of operations.

Chapter 3

Reservation Service - User Manual

1 Introduction

The Reservation Service allows clients to notify a manager how long they may require use of other services. A client may make a *reservation* for a specified period. Subsequently the reservation may be cancelled by requesting a reservation of zero interval. At any one time, there may be a number of client reservations.

The service manager may inspect the reservations whenever required. The manager may also set a *shutdown* time after which the availability of services in the distributed system is no longer guaranteed (for example, because of maintenance). If any client reservations are threatened by the shutdown time, the manager will be notified, and can then negotiate with the clients concerned or set a new shutdown time. Note that a client cannot make a reservation past the current shutdown time.

Normally a client will make a reservation for some reasonable period before using any other services. However a client may still use other services *without* making a reservation. In this case there is no guarantee about the availability of services.

2 Service state

A *reservation* records the user number (public identity) of the client who made it and the time at which it will expire. A number of reservations may exist at any one time. Each user may only have one reservation, and there is a limit on the total number of reservations.

$$\mathsf{Reservations} \mathrel{\widehat{=}} \{\, \mathsf{r} : \mathsf{UserNum} ⤀ \mathsf{Time} \mid \#\mathsf{r} \leq \mathit{Capacity} \,\}$$

The state of the Reservation Service records the shutdown time most recently set by the service manager (shutdown) and the set of current reservations (resns). The guest user cannot make reservations.

$$\begin{array}{|l}
\textsf{RS} \\
\hline
\quad \mathsf{shutdown} : \mathsf{Time} \\
\quad \mathsf{resns} \quad\;\; : \mathsf{Reservations} \\
\hline
\quad \mathit{GuestNum} \notin \mathsf{dom\ resns} \\
\hline
\end{array}$$

Initially the shutdown is set to a default value and there are no reservations.

$$\begin{array}{|l}
\textsf{InitRS} \\
\hline
\quad \mathsf{RS}' \\
\hline
\quad \mathsf{shutdown}' = \mathit{InitShutdownTime} \\
\quad \mathsf{resns}' = \emptyset \\
\hline
\end{array}$$

The service is in its initial state every time it is powered up.

3 Operation parameters

For each operation requested by clients there is an output parameter reporting the outcome of the operation (report!). Additionally the current time (now) and the user number of the client (clientnum) are available.

```
ΦBasicParams ________________________
| report!   : Report
| now       : Time
| clientnum : UserNum
|_____________________________________
```

Operations may change the state of the Reservation Service.

$$\Delta RS \ \widehat{=}\ RS \wedge RS' \wedge \Phi BasicParams$$

Some operations may leave the state of the service unchanged.

$$\Xi RS \ \widehat{=}\ \Delta RS \mid \theta RS = \theta RS'$$

Operations can return finite sets of users, so we make the following definition for the convenience of subsequent specifications.

$$Users \ \widehat{=}\ \{\ u : \mathbb{F}\ UserNum \mid \#u \leq \mathit{Capacity}\ \}$$

4 Reports

The report! output parameter of each operation indicates either that the operation succeeded or suggests why it failed.

Success indicates successful completion of an operation.

```
Success ______________________________
|   report! : Report
|---------------------
|   report! = SuccessReport
|______________________________________
```

If a reservation cannot be made due to early shutdown, the shutdown time itself is returned in until!. Note that a reservation of zero interval will not cause this error.

```
NotAvailable _________________________________
|   ≡RS
|   interval? : Interval
|   until!    : Time
|---------------------
|   interval? ≠ ZeroInterval
|   shutdown  < now + interval?
|   until!  = shutdown
|   report! = NotAvailableReport
|______________________________________________
```

The service has finite *capacity* for recording reservations; the report TooManyUsers occurs when that capacity would be exceeded. The report cannot occur if the client has a reservation (since it is overwritten by the new one).

```
TooManyUsers _________________________
|   ≡RS
|---------------------
|   #resns = Capacity
|   clientnum ∉ dom resns
|   report! = TooManyUsersReport
|______________________________________
```

Some operations can only be executed by the service manager.

```
NotManager ─────────────────────────
│   ≡RS
├──────────────
│   clientnum ≠ ManagerNum
│   report! = NotManagerReport
└───────────────────────────────────
```

Guest users cannot make reservations.

```
NotKnownUser ───────────────────────
│   ≡RS
├──────────────
│   clientnum = GuestNum
│   report! = NotKnownUserReport
└───────────────────────────────────
```

5 Service operations

Four operations are described in this section. Reserve, which may be performed by any authentic client, SetShutdown and Status, which may be performed only by the service manager, and Scavenge, which is performed by the service itself.

The description of each operation has three sections, titled Abstract, Definition and Reports.

The **Abstract** section gives a procedure heading for the operation, with formal parameters, as it might appear in some programming language. The correspondence between this procedure heading and an implementation of it in some real programming language is designed to be obvious and direct. A short informal description of the operation may accompany the procedure heading.

The **Definition** section mathematically defines the successful behaviour 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 subschemas (such as ΔRS). A short explanation may accompany the schema.

The **Reports** section summarises the report values which can be returned by the operation. This gives the definition of the total operation including the behaviour in the case of errors.

5.1 Client and manager operations

The following operations are available:

Reserve	—	make or clear a reservation
SetShutdown	—	set a shutdown time
Status	—	obtain current reservations

Only the Reserve operation may be performed by most clients. The last two operations may only be performed by the service manager.

RESERVE

Abstract

```
Reserve (interval? : Interval;
         until!    : Time;
         report!   : Report)
```

A reservation is made for a period of time (`interval?`), and returns the expiry time of the new reservation (`until!`).

A client can cancel a reservation by making a new reservation in which `interval?` is zero; this will then be removed by the next *scavenge*.

Definition

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathtt{Reserve}_{\mathtt{success}}} \\
\hline
\Delta\mathtt{RS} \\
\mathtt{interval?} : \mathtt{Interval} \\
\mathtt{until!} \quad : \mathtt{Time} \\
\hline
\mathtt{until!} \quad = \mathtt{now} + \mathtt{interval?} \\
\mathtt{shutdown'} = \mathtt{shutdown} \\
\mathtt{resns'} \quad = \mathtt{resns} \oplus \{\mathtt{clientnum} \mapsto \mathtt{until!}\} \\
\hline
\end{array}$$

Reports

$$\begin{array}{lll}
\mathtt{Reserve} & \mathrel{\widehat{=}} & (\mathtt{Reserve}_{\mathtt{success}} \land \mathtt{Success}) \\
 & & \oplus\ \mathtt{TooManyUsers} \\
 & & \oplus\ \mathtt{NotAvailable} \\
 & & \oplus\ \mathtt{NotKnownUser}
\end{array}$$

The client cannot be a guest user.

The reservation must expire before the shutdown time or be for a zero interval.

There may be no space for new reservations.

SETSHUTDOWN

Abstract

```
SetShutdown (shutdown?  : Time;
             threatens! : Users;
             report!    : Report)
```

The service manager may set a new shutdown time. All clients who have reservations which are threatened by the new time are returned. It is the responsibility of the service manager to negotiate with the clients affected.

Definition

```
SetShutdown_success ─────────────────────────────┐
│   ΔRS
│   shutdown?  : Time
│   threatens! : Users
├──────────────────
│   shutdown'  = shutdown?
│   resns'     = resns
│   threatens! =
│        dom (resns ▷ (ZeroTime..shutdown'))
└────────────────────────────────────────────────┘
```

The shutdown time is changed to the new value *regardless of existing reservations*. Current reservations are unaffected.

Clients with reservations past the new shutdown time are reported.

Reports

```
SetShutdown  ≙  (SetShutdown_success ∧ Success)
                                     ⊕ NotManager
```

This operation may only be performed by the service manager.

STATUS

Abstract

```
Status (now!      : Time;
        shutdown! : Time;
        resns!    : Reservations;
        report!   : Report)
```

The service manager may look at the current status of the service. The current and shutdown times are returned together with details of the current reservations.

Definition

```
┌─ Status_success ──────────────────────
│   ≡RS
│   now!      : Time
│   shutdown! : Time
│   resns!    : Reservations
├───────────────
│   now!      = now
│   shutdown! = shutdown
│   resns!    = resns
└───────────────────────────────────────
```

Reports

```
Status ≙ (Status_success ∧ Success)
                         ⊕ NotManager
```

This operation may only be performed by the service manager.

5.2 Internal operation - scavenging

In order to remove reservations which have expired, the service will perform a *scavenge* before each operation. This is in fact the *only* way in which reservations are removed.

```
Scavenge ________________________________________
|   RS
|   RS'
|   now : Time
|____________________
|
|   shutdown' = shutdown
|   resns'    = resns ⩥ (ZeroTime..now)
|________________________________________________
```

Scavenging does not change the shutdown time. A scavenge can remove reservations, but it never makes new ones. All reservations up to now are removed.

Performing a scavenge before each operation ensures that the service contains only non-expired reservations when the operation itself is performed. Thus, for example, the Status operation with only return current reservations.

6 Service charges

The basic parameters are supplemented by two *hidden* parameters, an operation identifier and the cost of executing the operation.

```
ΦParams ________________
|  ΦBasicParams
|  op?    : Op
|  cost!  : Money
|________________________
```

There is a fixed cost for each successful operation. All clients who make a reservation will also be charged an amount depending on the requested interval. There is no refund when a reservation is cleared to encourage clients to make reasonable requests.

```
RSTariff ____________________________________________________
|  ΦParams
|_______________________
|
|  op? = ReserveOp       ⟹ cost! = ReserveCost +
|                                 (TimeCost * interval?)
|  op? = SetShutdownOp   ⟹ cost! = SetShutdownCost
|  op? = StatusOp        ⟹ cost! = StatusCost
|_____________________________________________________________
```

where (_ * _) : (Money × Time) → Money is defined appropriately.

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

```
ErrorTariff  ≙  ΦParams | cost! = ErrorCost
```

These two schemas combine to form an overall tariff framing schema.

```
ΦRSTariff  ≙  Success ⟹ RSTariff     ∧
             ¬Success ⟹ ErrorTariff
```

7 Complete service

This section provides a definition of the complete reservation service. This uses schemas which are defined in the two previous sections as well as some schemas defined in the "Common Service Framework". Each operation is identified by an extra parameter op? which depends on the procedure name.

$$
\begin{array}{l}
\mathsf{RSServiceOps} \mathrel{\widehat{=}} \\
\quad \begin{array}{llll}
(\mathsf{Reserve} & \wedge\ \phi\mathsf{Params} & \mid\ \mathsf{op?} = \mathit{ReserveOp} &) \vee \\
(\mathsf{SetShutdown} & \wedge\ \phi\mathsf{Params} & \mid\ \mathsf{op?} = \mathit{SetShutdownOp} &) \vee \\
(\mathsf{Status} & \wedge\ \phi\mathsf{Params} & \mid\ \mathsf{op?} = \mathit{StatusOp} &)
\end{array}
\end{array}
$$

Each of these operations has a tariff associated with it, and they may all be considered to be preceded by an internal *scavenge* operation before the operation is invoked.

$$
\mathsf{RSBasicOps} \mathrel{\widehat{=}} \mathsf{Scavenge} \mathbin{;} (\phi\mathsf{RSTariff} \wedge \mathsf{RSServiceOps})
$$

The complete state and the initial state of the Reservation Service including a service clock, accounting and statistics as outlined in the "Common Service Framework" are:

$$
\mathsf{RSState} \mathrel{\widehat{=}} \mathsf{RS} \wedge \mathsf{Clock} \wedge \mathsf{Accts} \wedge \mathsf{Stats}
$$

$$
\mathsf{InitRSState} \mathrel{\widehat{=}} \mathsf{InitRS} \wedge \mathsf{InitClock} \wedge \mathsf{InitAccts} \wedge \mathsf{InitStats}
$$

The operations of the Reservation Service including a null operation and operations concerned with the service clock, accounting and statistics are as follows:

$$
\begin{array}{l}
\mathsf{RSAllOps} \mathrel{\widehat{=}} \\
\quad \begin{array}{llllll}
(\mathsf{BasicOps} & \wedge\ \Delta\mathsf{RS} & \wedge\ \phi\mathsf{Clock} & \wedge\ \phi\mathsf{Accts} & \wedge\ \phi\mathsf{Stats}) & \vee \\
(\mathsf{Null} & \wedge\ \equiv\mathsf{RS} & \wedge\ \phi\mathsf{Clock} & \wedge\ \phi\mathsf{Accts} & \wedge\ \phi\mathsf{Stats}) & \vee \\
(\mathsf{ClockOps} & \wedge\ \equiv\mathsf{RS} & \wedge\ \Delta\mathsf{Clock} & \wedge\ \phi\mathsf{Accts} & \wedge\ \phi\mathsf{Stats}) & \vee \\
(\mathsf{AcctsOps} & \wedge\ \equiv\mathsf{RS} & \wedge\ \phi\mathsf{Clock} & \wedge\ \Delta\mathsf{Accts} & \wedge\ \phi\mathsf{Stats}) & \vee \\
(\mathsf{StatsOps} & \wedge\ \equiv\mathsf{RS} & \wedge\ \phi\mathsf{Clock} & \wedge\ \equiv\mathsf{Accts} & \wedge\ \Delta\mathsf{Stats}) &
\end{array}
\end{array}
$$

Access control is not included since the shutdown time gives a form of access control.

Finally, the possibility of a bad operation and service error are included:

```
RSOps ≙
   ((BadOperation ∧ ≡SS ∧ ϕClock ∧ ≡Accts ∧ ≡Stats) ⊕
    (RSAllOps     ∧ ΔSS ∧ ΔClock ∧ ΔAccts ∧ ΔStats))
   ∨
   (ServiceError  ∧ ≡SS ∧ ϕClock ∧ ≡Accts ∧ ≡Stats)
```

Chapter 4

Reservation Service - Implementor Manual

1 Introduction

This document assumes that the reader is familiar with the "Reservation Service - User Manual" which outlines the *abstract* specification of the service. Here, this abstract specification is refined into a concrete specification of a possible implementation of the service. First the concrete state of the service is defined and then the concrete error and operation schemas are defined in terms of the concrete state components. Optimisations are included where this is desirable. The justification that the given concrete specification is a correct implementation of the abstract specification is discussed.

The specification given here is still not directly implementable. Predicates in schemas are given broadly in the order which the corresponding statements of a procedure in a sequential programming language might be written, as a hint to the implementor. A particular programming language must be chosen by the implementor and then this design must be refined into that language. Even with the advent of the use of formal specification in the design of computer based systems, it is anticipated that the job of the programmer is safe for some time to come.

2 Abstract state

The abstract state of the Reservation Service, as defined in the "Reservation Service - User Manual", includes the shutdown time most recently set by the service manager (shutdown) and the current reservations (resns). The guest user cannot make reservations.

$$\mathrm{Reservations} \mathrel{\widehat{=}} \{\, r : \mathrm{UserNum} \twoheadrightarrow \mathrm{Time} \mid \#r \leq \mathit{Capacity} \,\}$$

```
RS ________________________________
|   shutdown : Time
|   resns    : Reservations
|__________________
|   GuestNum ∉ dom resns
|__________________________________
```

Initially the shutdown time has an initial default value and there are no reservations.

```
InitRS ____________________________
|   RS'
|__________________
|   shutdown' = InitShutdownTime
|   resns'    = ∅
|__________________________________
```

Full details of the abstract operations on the service can be found in the User Manual.

3 Concrete state

In the abstract state, the reservations are modelled as a partial function. We shall assume that the number of clients with reservations at any particular time is relatively small compared to the total number of clients (i.e. the function is sparse).

Hence in the concrete state, we shall implement this partial function as a pair of arrays containing matching user numbers and reservation times at corresponding array indices. Since not all entries in these arrays need be in use at any given moment, we need a special user number to indicate an empty entry. The guest user is not allowed to make reservations, and cannot appear as a user number in the reservation table, so we shall therefore use this number to denote unused entries in the array.

$$\begin{array}{|l}
\mathit{Unused} : \mathsf{UserNum} \\
\hline
\mathit{Unused} = \mathit{GuestNum}
\end{array}$$

The arrays have indices limited to a maximum upper bound *Capacity* which determines the number of clients for whom the service can hold reservations simultaneously. This limit must be determined by the implementor according to the estimated usage of the service.

$$\mathsf{Index} \;\widehat{=}\; 1 \,..\, \mathit{Capacity}$$

$$\begin{array}{lll}
\mathsf{UserArray} & \widehat{=} & \mathsf{Index} \rightarrow \mathsf{UserNum} \\
\mathsf{TimeArray} & \widehat{=} & \mathsf{Index} \rightarrow \mathsf{Time}
\end{array}$$

The shutdown time may easily be implemented as a single variable (shutd), so that the concrete implemented service state consists of three components.

$$\begin{array}{|l}
\hline
\mathsf{cRS} \\
\hline
\mathsf{shutd} : \mathsf{Time} \\
\mathsf{users} : \mathsf{UserArray} \\
\mathsf{times} : \mathsf{TimeArray} \\
\hline
(\mathsf{users} \rhd \{\mathit{Unused}\}) \in (\mathsf{Index} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathsf{UserNum}) \\
\hline
\end{array}$$

Each authentic client can have at most one entry in the users array, all other entries being unused.

Initially the shutdown time has the default value and all the entries in the user array are unused. (It will not matter what values are held in the time array.)

```
cInitRS ________________________
| cRS'
|_______________
| shutd' = InitShutdownTime
| users' = λ s:Index • Unused
|________________________________
```

For each operation requested by clients there is an indication of the outcome of the operation (report!). Additionally the current time (now) and the user number of the client (clientnum) are available.

```
ΦBasicParams ____________________
| report!   : Report
| now       : Time
| clientnum : UserNum
|________________________________
```

Operations may change the state of the Reservation Service implementation.

$$\Delta cRS \ \widehat{=}\ cRS \wedge cRS' \wedge \Phi BasicParams$$

Some operations may leave the state of the service unchanged.

$$\Xi cRS \ \widehat{=}\ \Delta cRS \mid cRS = cRS'$$

Operations can return finite sets of users, so we make the following definition for the convenience of subsequent specifications.

$$Users \ \widehat{=}\ \{\ u : \mathbb{F}\ UserNum \mid \#u \leq Capacity\ \}$$

4 Reports

The report schema definitions are little changed in the implementation because they mainly do not involve refined state components.

```
┌ cSuccess ──────────────────────────────
│ ΔcRS
├──────────
│ report! = SuccessReport
└────────────────────────────────────────
```

```
┌ cNotManager ───────────────────────────
│ ΞcRS
├──────────
│ clientnum ≠ ManagerNum
│ report!   = NotManagerReport
└────────────────────────────────────────
```

```
┌ cNotKnownUser ─────────────────────────
│ ΞcRS
├──────────
│ clientnum = GuestNum
│ report!   = NotKnownUserReport
└────────────────────────────────────────
```

```
┌ cNotAvailable ─────────────────────────
│ ΞcRS
│ interval? : Interval
│ until!    : Time
├──────────
│ interval? ≠ ZeroInterval
│ shutd     < now + interval?
│ until!    = shutd
│ report!   = NotAvailableReport
└────────────────────────────────────────
```

(The TooManyUsers report schema has been directly incorporated in the implementation of the Reserve operation.)

5 Operation implementations

The four service operations are redefined here in terms of the refined concrete state. As in the "User Manual", the description of each operation has three sections, titled Abstract, Definition and Reports.

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 the client's 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. Extra state components indicate extra variables which will be required in the final program.

The **Reports** sections covers error conditions to produce a formal description of the total operation.

Each schema definition may be conveniently implemented as a procedure in the final program.

RESERVE

Abstract

```
Reserve (interval? : Interval;
         until!    : Time;
         report!   : Report)
```

Definition

In the concrete form of this operation, the combination of the Success and TooManyUsers report cases are optimised into a combined 'available' definition.

```
cReserve_avail ──────────────────────────────
  ΔcRS
  interval? : Interval
  until!    : Time
  i, j      : Index
 ──────────
  shutd' = shutd
  until! = now + interval?
  clientnum ∈ ran users ⇒
      users i = clientnum
      users' = users
      times' = times ⊕ {i ↦ until!}
      report! = SuccessReport
  clientnum ∉ ran users ⇒
      Unused ∈ ran users ⇒
          users j = Unused
          users' = users ⊕ {j ↦ clientnum}
          times' = times ⊕ {j ↦ until!}
          report! = SuccessReport
      Unused ∉ ran users ⇒
          users' = users
          times' = times
          report! = TooManyUsersReport
 ────────────────────────────────────────────
```

The shutdown time is unaffected.

A check is made to see whether an entry for the client already exists in the users array. If a client already has a reservation entry, then that entry in the array (with index i) is used. Otherwise, if there are any unused entries in the array, one of them (with index j) is used.

If the client does not have an existing entry and there are no unused entries, the state remains unchanged and an error report is given.

Reports

$$
\begin{aligned}
cReserve \mathrel{\widehat{=}}\ & cReserve_{avail} \\
& \oplus cNotAvailable \\
& \oplus cNotKnownUser
\end{aligned}
$$

SETSHUTDOWN

Abstract

```
SetShutdown (shutdown?  : Time;
             threatens! : Users;
             report!    : Report)
```

Definition

$$
\begin{array}{|l}
\hline
\textsf{cSetShutdown}_{\textsf{success}} \\
\quad \Delta\textsf{cRS} \\
\quad \textsf{shutdown?} : \textsf{Time} \\
\quad \textsf{threatens!} : \textsf{Users} \\
\hline
\quad \textsf{shutd}' = \textsf{shutdown?} \\
\quad \textsf{users}' = \textsf{users} \\
\quad \textsf{times}' = \textsf{times} \\
\quad \textsf{threatens!} = \\
\qquad \{\ \textsf{i}:\textsf{Index} \mid (\textsf{users i} \neq \mathit{Unused}) \wedge \\
\qquad\qquad\qquad (\textsf{times i} > \textsf{shutd}') \bullet \textsf{users i}\ \} \\
\hline
\end{array}
$$

The shutdown time is set but the arrays are left unaffected.

The set of threatened users is returned. Each such user must have a valid entry in the user array and a reservation time past the new shutdown time.

Reports

$$
\begin{array}{lll}
\textsf{cSetShutdown} & \widehat{=} & (\textsf{cSetShutdown}_{\textsf{success}} \wedge \textsf{cSuccess}) \\
 & & \quad \oplus\ \textsf{cNotManager}
\end{array}
$$

STATUS

Abstract

```
Status (now!      : Time;
        shutdown! : Time;
        resns!    : Reservations;
        report!   : Report)
```

Definition

```
cStatus_success ──────────────────────────────
│   ΞcRS
│   now!      : Time;
│   shutdown! : Time;
│   resns!    : Reservations;
├─────────────────
│   now!      = now
│   shutdown! = shutd
│   resns!    = { i:Index | users i ≠ Unused •
│                          (users i ↦ times i) }
└──────────────────────────────────────────────
```

The state of the service is not changed.

All the valid user array entries and their corresponding reservation times are returned as a set of pairs. Threatened reservations may be deduced by the calling program from the shutdown time.

Reports

```
cStatus ≙ (cStatus_success ∧ cSuccess)
                           ⊕ cNotManager
```

Scavenging

In order to remove reservations which have expired, the service will perform a *scavenge* before each operation. This is in fact the *only* way in which reservations are removed.

```
cScavenge ________________________________________
|   cRS
|   cRS′
|   now : Time
|_______________
|   shutd′ = shutd
|   ∀ i:Index •
|       (users i = Unused) ∨ (times i < now)  ⇒
|           users′ i = Unused
|       (users i ≠ Unused) ∧ (times i ≥ now)  ⇒
|           users′ i = users i
|   times′ = times
|__________________________________________________
```

Scavenge does not change the shutdown time.

Valid entries with reservation times in the past are removed from the user array. The reservation time array is left unchanged. The entries in the time array corresponding to unused entries in the user array may be ignored.

6 Complete service

This section provides a combined definition of the operations of the implemented Reservation Service. It does not include details of the implementation of service components, such as accounting and statistics, 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 on the refined state of the service, the definition of the ΦRSTariff 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:

```
cRSServiceOps ≙
    (cReserve      ∧ ΦParams | op? = ReserveOp     ) ∨
    (cSetShutdown  ∧ ΦParams | op? = SetShutdownOp ) ∨
    (cStatus       ∧ ΦParams | op? = StatusOp      )

cRSBasicOps ≙ cScavenge ⨾ (ΦRSTariff ∧ cRSServiceOps)
```

7 Implementation correctness

It is necessary to show that the implementation of the Reservation Service as described in this manual correctly implements the view presented in the user manual. In order to do this, the state refinement step is expressed as an *abstraction relation*, and service initialisation and each of the operation implementations must be shown to achieve the appropriate change of state with respect to this relation.

For service initialisation, the concrete initial state must be shown to lead 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.

When a complete definition is constructed by composing a number of schemas, such as in defining the error behaviour of an operation, the proof can be constructed in an equivalent manner.

The rest of this section describes what needs to be proved in order to show the correctness of the implementation. The proofs themselves, because of their length, are omitted here but are contained in [8].

State refinement

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

$$
\begin{array}{|l}
\text{RelRS} \\
\hline
\quad \text{RS} \\
\quad \text{cRS} \\
\hline
\quad \text{shutdown} = \text{shutd} \\
\quad \text{resns} = (\text{users} \mathbin{⩥} \{\mathit{Unused}\})^{-1} \mathbin{⨾} \text{times} \\
\hline
\end{array}
$$

The abstract variable shutdown is exactly implemented by the concrete variable shutd. The abstract reservations are found by taking the time entries in the concrete time array which correspond to each of the 'used' entries in the concrete user array.

To show that there is a concrete implementation for every abstract state, we must prove that

$$\vdash \ \forall\, RS \cdot \exists\, cRS \cdot RelRS$$

Initialisation

To show the correctness of the service initialisation we must prove that

$$cInitRS \ \vdash \ \exists\, RS' \cdot InitRS \wedge RelRS'$$

Operation implementations

First consider the Reserve operation. In order to show the correctness of the total operation, we first show the correctness of the partial operation in the 'available' case. This corresponds to $cReserve_{avail}$ in the implementation and the following in the user manual.

$$Reserve_{avail} \ \widehat{=} \ (Reserve_{success} \wedge Success) \oplus TooManyUsers$$

To show that the partial concrete operation is as applicable as the partial abstract one we must prove that

$$pre\ Reserve_{avail} \wedge RelRS \ \vdash \ pre\ cReserve_{avail}$$

To show that the partial concrete operation correctly implements the partial abstract operation we must prove that

$$\begin{array}{l} pre\ Reserve_{avail} \wedge cReserve_{avail} \wedge RelRS \\ \quad \vdash \ \exists\, RS' \cdot Reserve_{avail} \wedge RelRS' \end{array}$$

In order to demonstrate the correctness of the total operation, we must extend the 'available' case to include the other possible error conditions. The total concrete and abstract definitions are respectively

$$\begin{array}{rcl} cReserve & \widehat{=} & cReserve_{avail} \oplus cNotAvailable \oplus cNotKnownUser \\ Reserve & \widehat{=} & Reserve_{avail} \oplus NotAvailable \oplus NotKnownUser \end{array}$$

In both of the additional error cases, the error schemas depend only on the operation parameters and not on the service state (in either concrete or abstract form). They both leave the service state unchanged. Hence the refinement of the state does not change either the applicability or the correctness of the total operations.

The correctness of the SetShutDown and Status operations follow the same pattern of showing the correctness of the partial operation in the 'success' case and then extending it to the total case.

Chapter 5

Discussion and Experience

1 General

This chapter discusses some of the achievements and of the second phase of the project. Some changes in the style of specifications contained in the user manuals have been made. Implementor manuals have been provided in the same formal style. Additionally, the Common Service Framework has been developed to simplify service specification. Experience has been gained using the original services, and this has increased our confidence in the specifications presented.

1.1 User manual format

The style of the User Manuals has been improved during the second phase of the project. For example, the description of the Reservation Service from the first phase, presented in [1] and [7], can be compared with that presented here.

Error conditions have been more exactly specified in the Reports section for each operation, using the schema overriding operator ($\oplus$) to define an order of checking for error conditions.

The cost of performing operations has been gathered together in a tariff schema after the operations themselves have been presented. The cost of an operation is often of secondary interest to understanding *what* the operation does, and clutters its specification.

At the end of each manual, the operations specific to the service are combined with those incorporated from the Common Service Framework to produce an overall specification of the operations available in the service.

The initial state is now included formally for each service. The state of a service at any given time is the result of the initial state being composed with *all* the operations which have been performed to date.

1.2 Service implementation

With the introduction of Implementor Manuals, it has been possible to present an implementor's view of a service, showing how the abstract user's view can be refined towards a concrete implementation.

A significant amount of effort has been spent on the presentation of these manuals, since it is all too easy for them to become swamped by detail. The implementor manual for the Reservation Service, included here, is a relatively straightforward example because of the simplicity of the service itself.

It has not been possible in the time available on this project to take the refinement of the implementation of every service all the way down to the code of a particular programming language. Other work in progress at Oxford has considered this step in more detail [9]. We have concentrated on the 'architectural' aspects of system design, taking a top-down approach in which the structure of the implementation has been of greatest concern.

1.3 Representation of parameters

The types of parameters of service operations have been presented as Z sets. These can be either given sets, assumed to be unstructured, such as Time or Report, or they can be defined in Z as a set, sequence or other more complicated structure.

We have ignored the issue of how such types will be represented in a specific programming language. Clearly, at the lowest level, the parameter values must be transmitted over the network between client and service in some bit pattern. Since there is no assumption that all client applications and service code will be written in the same programming language, there would need to be a clear specification of the representation at this level so that data conversion functions could be applied if necessary.

Take, as an example, the set of Reservations which is returned by the Status operation of the Reservation Service. This consists of a partial function (of limited size) from UserNum to Time. Most programming languages would not be able to implement this directly. Typically it could be implemented as an array with elements consisting of a record containing a user number and associated time. The ordering of the array could be arbitrary, or it may be ordered by user number or time.

Parameter refinement is still a topic under active discussion. It could be considered as a relation between abstract and concrete parameters in a similar manner to the way abstract and concrete states are related. It would therefore form a second, orthogonal, dimension of refinement to that of the implementation of a service.

2 Common framework

The introduction of the Common Service Framework has allowed a number of definitions common to several services to be grouped together in one document. This has also meant that the specifications of individual services have been made that much simpler.

The specification of the common framework has also illustrated how separate subsystems can be defined, with their own state and operations, and then incorporated into the definition of a complete service. It has addressed, at the specification level, the issues of errors in the implementation of services or in the network over which they are accessed.

An Implementor Manual should be provided for the Common Service Framework, which refines the state and provides operation implementations for each of the subsystems and other components introduced in the common framework. This should be a straightforward exercise, following the same pattern as the manuals provided for other services.

2.1 Service and network errors

There are two kinds of errors specified in the common framework which are non-deterministic. In other words, they do not arise because of some predicate which the client's parameters have failed to satisfy, but because of an error arising in the underlying implementation. Service errors are caused by a failure in the service implementation, such as a disk error in a storage service. Network errors are caused by a failure in communication over the network.

Both kinds of error have been made visible to the client through the return of corresponding error report values. It is left to the client's application to take appropriate action in the case of such errors arising. At a higher level of abstraction, it might be possible to hide transient errors from the client by automatically retrying operations until they achieved a definite result (i.e success, or a specific error report).

The specification of these non-deterministic errors is a problem. When a service error occurs, we have specified that the state of the service remains unchanged. This may be hard to achieve in practice. For example, if a disk crashes and loses some of its data, the service will clearly not be able to maintain that part of its state. To keep within its specification, it would be obliged to return a service error for any subsequent operation which depended on information in the lost part of the state, effectively rendering it invisible to any client.

When a network error occurs, we have specified that either the state of the service remains unchanged or that the operation has been completed (though the result is not visible to the client). These two cases correspond to a communication failure in transmitting the operation request or reply respectively. On receiving such an error report, the client may re-attempt the operation. However, if the operation is not idempotent, such as one which creates or deletes a component in the service state, this will produce unwanted side-effects. A stricter specification might eliminate the second case, so that this error could be handled in the same way as a service error. The network implementation would then be obliged to provide a mechanism to recover from loss of operation replies.

2.2 Operators on basic sets

One area which is of concern in many Z specifications involves dealing with the partial nature of some of the underlying operators.

Operators such as addition, subtraction and comparison are assumed to exist for some of the sets, such as Time and Money, introduced in the Common Service Framework. These operators are defined to be total in the abstract specification to avoid having to introduce error checks and reports when they are invoked outside their domain.

Since these sets are to be implemented they must be finite. Hence 'overflow' or 'underflow' (i.e the required result lies outside the defined range) could occur when adding or subtracting some values. Many arithmetic implementations in hardware simply wrap round when this occurs, producing undetectable invalid results. A more sensible approach is to return some standard error value in these cases. Output parameters may be checked for this value by the client if desired.

3 Reservation Service

The following comments refer specifically to the Reservation Service presented in the previous chapters.

3.1 Design of the service

The design of the Reservation Service reflects its provision as part of a distributed system rather than a monolithic system. When building a distributed operating system from a number of services which are largely independent of each other, it is possible that one service needs to be enabled or disabled while other services and clients continue to run.

When a service is disabled (*shutdown*), there should not be any client who is at that moment involved in some *series* of interactions with it - because interruption of such a series could be quite inconvenient for the client. If these series (or *transactions*) can be recognised by the service, it is possible to avoid this inconvenience as follows.

Possible shutdown procedure:

1. The service manager *requests* shutdown of the service.
2. The service rejects any attempt to begin a new transaction, but allows current transactions to continue.
3. When all transactions have completed, the service notifies the service manager that shutdown is complete.

However, there are some problems; for example, a client might fail to complete a transaction (presumably due to accidental failure of the client's own software). If this happened, the service would *never* shutdown. A second problem is that for some services (*e.g.* the low-level block storage service) there is no recognisable transaction structure, and so the above scheme cannot be used at all.

The Reservation Service presented here does not interact at all with the service or services it reserves; it interacts only with its own clients, and with the service manager. It allows clients to state for how long they would like to use the available services, and it allows the service manager to set a *shutdown time* beyond which all reservations are to be rejected. It becomes the clients' responsibility to protect themselves from sudden shutdown of the services (by making reservations), and the service manager's responsibility to disable the service only after the shutdown time. Thus a shutdown can be unexpected only by those clients who have made no reservation (or if the service manager deliberately ignores outstanding reservations).

A typical use of the Reservation Service would be for clients to include a reservation request at the start of every program using the available services. The duration of the reservation should be long enough to allow the program to complete, but short enough to allow the service manager to make a reasonably spontaneous decision to shutdown. A reservation time of half an hour has proved convenient in practice for many of the applications making use of our services.

3.2 A problem discovered

The Reservation Service was in use before the start of the current phase of the project, and its original User Manual has been published previously [1,7]. However an error was discovered during the use of the service which was not anticipated during the design stage. This has led to a small revision in the specification of one of the error schemas for the service.

The problem arose when a client made a reservation successfully and subsequently tried to clear it by making a reservation of zero-interval in the normal way. However the service reported that it was "*Not Available*" and hence the reservation could not be removed.

The specification in the User Manual was examined to see how this state of affairs might transpire. To obtain the "*Not Available*" report, the following precondition in the NotAvailable schema had to hold:

```
        shutdown < now + interval?
```

With interval? being zero, this implied that the shutdown time was set earlier than the current time. Given that the client had earlier successfully made a reservation that was still in force (and hence needed to be cleared), this implied that the shutdown time had been brought forward by the service manager, threatening the pending reservation. In fact, the client was a laser printing service which was known to always make half-hour reservations. The manager had set a shutdown time earlier than the end of the printing service's reservation time, assuming that it could clear its current reservation but not make any new reservations. The manager was prepared to wait until the reservation was cleared as an indication that the printer had finished its current job – but the reservation was never cleared.

3.3 A problem solved

To prevent the problem of not being able to cancel reservations after the shutdown time, two solutions were proposed. The choice between them illustrates the kind of design choice in which additional complexity in a single operation may be balanced against the use of an additional operation. A first solution involves adding an extra precondition to the original NotAvailable schema:

$$interval? \neq ZeroInterval$$

This means that a call to the Reserve operation with a *ZeroInterval* can no longer return with a *NotAvailableReport*.

This is the solution presented in the previous chapters in which the Reserve operation serves the dual purpose of making a reservation (when $interval? \neq ZeroInterval$) and also clearing a reservation (when $interval? = ZeroInterval$).

An alternative solution to this would be to provide a new Cancel operation for clearing a reservation. This complicates the service by providing an extra operation, which is the reason it was not included in the original version of the service. However it is likely that its inclusion would have prevented the problem just described from arising. A specification for this operation is presented overleaf. Note that the NotAvailable schema need no longer check for a *ZeroInterval*. A client could still make a reservation for a zero interval, which would normally clear the reservation except in the circumstances described above.

An additional error schema is required for use with the Cancel operation. This returns an error report if the client has no outstanding reservation.

```
NotReserved ________________________________
|    ΞRS
|___________________
|
|    clientnum ∉ dom resns
|    report! = NotReservedReport
|___________________________________________
```

CANCEL

Abstract

$$\mathsf{Cancel}\ (\mathsf{report!} : \mathsf{Report})$$

A client can cancel a reservation which has previously been made.

Definition

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathsf{Cancel}_{\mathsf{success}}} \\
\quad \Delta \mathsf{RS} \\
\hline
\quad \mathsf{shutdown}' = \mathsf{shutdown} \\
\quad \mathsf{resns}' \quad\ = \{\mathsf{clientnum}\} ⩤ \mathsf{resns} \\
\hline
\end{array}$$

The shutdown time is unaffected. The client's reservation is removed.

Reports

$$\begin{aligned}\mathsf{Cancel} \quad \widehat{=} \quad & (\mathsf{Cancel}_{\mathsf{success}} \wedge \mathsf{Success}) \\ & \qquad\qquad \oplus\ \mathsf{NotReserved}\end{aligned}$$

An error is reported if the client does not have an outstanding reservation.

3.4 Reservation of individual services

As implemented, the Reservation Service is a separate service in its own right. Making a reservation does not affect any other service, and it is assumed that the client may wish to make use of any service. However in a larger distributed system, with a greater number of services to choose from, it may well make sense to include the reservation operations in individual services so that they may be shutdown independently. To do this, another layer would need to be defined in the "Common Service Framework" containing the reservation state and operations. This could then be included in any services requiring their own (standard) reservation and shutdown procedures.

3.5 Proof of correctness

The design in the Implementor Manual has been proven correct with respect to the User Manual. Due to their length, the proofs have been omitted here; they can be found in a separate document [8]. In addition, the document shows how the operations can then be programmed in Dijkstra's guarded command language to meet the specifications in the Implementor Manual.

Acknowledgements

Thank you to the developers of the Z specification language at the PRG for providing the environment to make this monograph possible. Carroll Morgan and Tim Gleeson, both previous members of the project, provided both inspiration in the style of presentation and perspiration in the production of service manuals and software. They and Bernard Sufrin made useful comments on later drafts of this monograph.

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. Topp-Jørgensen, S. "Reservation Service: Implementation Correctness Proofs", DCS project working paper, *Programming Research Group, Oxford University*, (1987).

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

Appendix A

Index of formal definitions

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

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.

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

2. Logic.

Let P, Q be predicates and D declarations.

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

3. Sets.

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

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

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

4. Relations.

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

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

5. Functions.

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

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

$X ⤔ Y$	The set of partial injective (one-to-one) functions from X to Y: $\widehat{=} \{f : X ⇸ Y \mid \forall y : \mathrm{ran}\ f \bullet (\exists_1 x : X \bullet f\ x = y)\}$.
$X ↣ Y$	The set of total injective functions from X to Y: $\widehat{=}\ (X ⤔ Y) \cap (X \rightarrow Y)$.
$X ⤀ Y$	The set of partial surjective functions from X to Y: $\widehat{=} \{f : X ⇸ Y \mid \mathrm{ran}\ f = Y\}$.
$X \twoheadrightarrow Y$	The set of total surjective functions from X to Y: $\widehat{=}\ (X ⤀ Y) \cap (X \rightarrow Y)$.
$X ⤖ Y$	The set of total bijective (injective and surjective) functions from X to Y: $\widehat{=}\ (X \twoheadrightarrow Y) \cap (X ↣ Y)$.
$X ⇻ Y$	The set of finite partial functions from X to Y: $\widehat{=} \{f : X ⇸ Y \mid f \in \mathbb{F}\ (X \times Y)\}$.
$⇸ ⤔ ⤀ ⤗$	Partial functions.
$\rightarrow ↣ \twoheadrightarrow ⤖$	Total functions.
$⇻ ⤕ ⤀ ⤘$	Finite functions.
$f_1 \oplus f_2$	Functional overriding: given $f_1, f_2 : X ⇸ Y$, $\widehat{=}\ (\mathrm{dom}\ f_2 ⩤ f_1) \cup f_2$.
$f\ \_$	Prefix function declaration (default if no underlines used).
$(\_\ f\ \_)$	Infix function declaration (often underlined in use for clarity).
$\_\ f$	Postfix function declaration.
$f\ t$	The function f applied to t.
$f(t)$	$\widehat{=}\ f\ t$.
$\lambda D \mid P \bullet t$	Lambda-abstraction: the function that, given an argument x of type X such that P holds, the result is t. Given $D \widehat{=} x_1 : T_1; \ldots ; x_n : T_n$, $\widehat{=} \{D \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$.
$\lambda D \bullet t$	$\widehat{=}\ \lambda D \mid \mathrm{true} \bullet t$

6. Numbers.

Let m, n be natural numbers.

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

7. Orders.

$\mathrm{partial\_order}\ X$

The set of partial orders on X:

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

$\mathsf{total\_order}\ X$
The set of total orders on X:
$\widehat{=}\ \{R : \mathsf{partial\_order} \mid \forall x, y : X \bullet x\,R\,y \lor y\,R\,x\}$.

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

8. Sequences.

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

$\mathsf{seq}\ X$ The set of sequences whose elements are drawn from X:
$\widehat{=}\ \{A : \mathbb{N} \twoheadrightarrow X \mid \mathrm{dom}\ A = 1..\#A\}$.

$\langle\rangle$ The empty sequence $\emptyset$.

$\mathsf{seq}_1\ X$ The set of non-empty sequences:
$\widehat{=}\ \mathsf{seq}\ X \setminus \{\langle\rangle\}$

$\langle a_1, \ldots, a_n\rangle$
$\widehat{=}\ \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$.

$\langle a_1, \ldots, a_n\rangle \frown \langle b_1, \ldots, b_m\rangle$
Concatenation:
$\widehat{=}\ \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$,
$\langle\rangle \frown A = A \frown \langle\rangle = A$.

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

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

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

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

$\mathsf{rev}\ \langle a_1, a_2, \ldots, a_n\rangle$ Reverse:
$\widehat{=}\ \langle a_n, \ldots, a_2, a_1\rangle$,
$\mathsf{rev}\ \langle\rangle = \langle\rangle$.

$\frown/AA$ Distributed concatenation:
given $AA : \mathsf{seq}(\mathsf{seq}(X))$,
$\widehat{=}\ AA(1) \frown \ldots \frown AA(\#AA)$,
$\frown/\langle\rangle = \langle\rangle$.

$;/AR$ Distributed relational composition:
given $AR : \mathsf{seq}\ (X \leftrightarrow X)$,
$\widehat{=}\ AR(1)\ ;\ \ldots\ ;\ AR(\#AR)$,
$;/\langle\rangle = \mathrm{id}\ X$.

$\oplus/AR$ Distributed overriding:
given $A : \mathsf{seq}\ (X \twoheadrightarrow Y)$,
$\widehat{=}\ AR(1) \oplus \ldots \oplus AR(\#AR)$,
$\oplus/\langle\rangle = \emptyset$.

$\mathsf{squash}\ f$ Convert a finite function, $f : \mathbb{N} \twoheadrightarrow X$, into a sequence by squashing its domain. That is,
$\mathsf{squash}\ \emptyset = \langle\rangle$,
and if $f \neq \emptyset$ then
$\mathsf{squash}\ f = \langle f(i)\rangle \frown \mathsf{squash}(\{i\} \ntriangleleft f)$
where $i = \min(\mathrm{dom}\ f)$.

$S \upharpoonleft A$ Index restriction:
$\widehat{=}\ \mathsf{squash}(S \lhd A)$.

$A \upharpoonright T$ Sequence restriction:
$\widehat{=}\ \mathsf{squash}(A \rhd T)$.

$\mathsf{disjoint}\ AS$ Pairwise disjoint:
given $AS : \mathsf{seq}\ (\mathbb{P}\ X)$,
$\widehat{=}\ (\forall i, j : \mathrm{dom}\ AS \bullet i \neq j \Rightarrow AS(i) \cap AS(j) = \emptyset)$.

$AS\ \underline{\mathsf{partitions}}\ S$
$\widehat{=}\ \mathsf{disjoint}\ AS \land \bigcup \mathrm{ran}\ AS = S$.

$A\ \underline{\mathsf{in}}\ B$ Contiguous subsequence:
$\widehat{=}\ (\exists C, D : \mathsf{seq}\ X \bullet C \frown A \frown D = B)$.

9. Bags.

$\mathsf{bag}\ X$ The set of bags whose elements are drawn from X: $\widehat{=}\ X \twoheadrightarrow \mathbb{N}_1$

$\mathsf{items}\ s$ The bag of items contained in the sequence s: $\widehat{=}\ \{x : \mathrm{ran}\ s \bullet x \mapsto \#\{i : \mathrm{dom}\ s \mid s(i) = x\}\}$

Schema Notation

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

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

or horizontally, for the same example

S ≙ [x: N; y: seq N | x≤#y].

Use in signatures after ∀, λ, {...}, etc.:

(∀S • y ≠ ⟨⟩) ≙ (∀x:N; y: seq N | x≤#y • y≠⟨⟩).

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 seq N) such that x ≤ #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 ≤ #w.y. Additionally, given w : X ⇸ S, w ⨾ (λS.x) is a function X ⇸ 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 ≤ #y.

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

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

is

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

S | P The schema S with P conjoined to its predicate part. E.g., (S | x>0) is [x:N; y:seq N | x≤#y ∧ 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:seq N | x≤#y].

S[new/old] Renaming of components: the schema S in which the component old has been renamed to new both in the declaration and at its every free occurrence in the predicate. For example, S[z/x] is [z:N; y:seq N | z ≤ #y] and S[y/x, x/y] is [y:N; x:seq N | y ≤ #x].

In the second case above, the renaming is simultaneous.

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

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

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

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

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

```
┌──────────────────────
│ x : N
│ y : seq N
│ z : P N
├──────────────────
│ x ≤ #y ∨ x ∈ z
└──────────────────────
```

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

```
┌──────────────────────
│ x : N
│ y : seq N
│ z : P N
├──────────────────
│ x ≤ #y ⇒ x ∈ z
└──────────────────────
```

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

```
┌──────────────────────
│ x : N
│ y : seq N
│ z : P N
├──────────────────
│ x ≤ #y ⇔ x ∈ z
└──────────────────────
```

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

Hiding: the schema S with the variables v_1, $v_2, \ldots$, and v_n hidden: the variables listed are removed from the declarations and are existentially quantified in the predicate. E.g., $S \setminus x$ is $[\,y:\mathrm{seq}\ \mathbb{N} \mid (\exists x:\mathbb{N} \cdot x \leq \#y)\,]$. (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, $(S \land T) \setminus S$ is

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

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

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

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

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

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

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

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

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

pre S is

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

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

S ⊕ T
Overriding:
≙ (S ∧ ¬pre T) ∨ T.
For example, given S above and

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

S ⊕ T is

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

which simplifies to

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

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

```
┌───────────────────────────
│ x?, s, s′, y! : ℕ
├─────────────────────
│ (∃ s₀ : ℕ .
│   s₀ = s−x ∧ y! = s ∧
│   s₀ < x? ∧ s′ = s₀)
└───────────────────────────
```

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

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

The following conventions are used for prefixing of schema names:

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

For example

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

Other Definitions

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

```
│ declaration(s)
├─────────────────────
│ predicate(s)
```

or horizontally: $D \mid P$

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

```
╔═[A, B, ...]═══════════════
│ declaration(s)
├─────────────────────
│ predicate(s)
└───────────────────────────
```

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

```
┌─S[A, B, ...]──────────────
│ declaration(s)
├─────────────────────
│ predicate(s)
└───────────────────────────
```