SPECIFYING THE CICS APPLICATION PROGRAMMER'S INTERFACE

Ian Hayes

Technical Monograph PRG-47

July 1985

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

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

Author's address from September 1985:
Department of Computing Science
Queensland University
St. Lucia
Queensland 4067
Australia

SPECIFYING THE CICS APPLICATION PROGRAMMER'S INTERFACE

Ian Hayes

Contents

Preface

This monograph contains papers produced as part of a joint project between IBM (UK) Laboratories at Hursley, England and the Programming Research Group of Oxford University into the application of formal software specification techniques to industrial problems. The work documented consists of specification of parts of the IBM Customer Information Control System (CICS).

The first paper contains a description of the work carried out; this paper has been published in the IEEE Transactions on Software Engineering (February 1985). A number of modules of the CICS command level application programmer's interface were specified; these include the CICS Exception Handling which is documented in the first paper and CICS Temporary Storage and Interval Control which are described in separate papers. The paper on the CICS Message System was later work not directly related to the other papers.

The work documented here was supported by research contract between IBM and Oxford University and is published by kind permission of the Company.

APPLYING FORMAL SPECIFICATION TO SOFTWARE DEVELOPMENT IN INDUSTRY*

ABSTRACT

This paper reports experience gained in applying formal specification techniques to an existing transaction processing system. The system is the IBM Customer Information Control System (CICS) and the work has concentrated on specifying a number of modules of the CICS application programmer's interface.

The uses of formal specification techniques with particular reference to their application to an existing piece of software are outlined. The specification process itself is described and a sample specification presented. The specifications are written in the specification notation Z, which is based on the notation of set theory from mathematics.

One of the main benefits of applying specification techniques to existing software is the questions that are raised about the system design and documentation during the specification process. Some samples of the problems that were identified by these questions are discussed.

Problems with the specification techniques themselves, that were encountered in applying the techniques to a commercial transaction processing system are outlined.

* This paper is a minor revision of the paper of the same title published in the IEEE Transactions on Software Engineering, volume 11, number 2 (February, 1985).

INTRODUCTION

Oxford University and IBM United Kingdom Laboratories Limited are engaged in a joint project to evaluate the applicability of formal specification techniques to industrial scale software. The project is attempting to scale up formal mathematical methods so far used within a research environment to large scale software in an industrial environment. This paper reports the experience gained so far in applying these techniques to describe the application programmer's interface of the IBM Customer Information Control System (CICS).

CICS is widely used to support online transaction processing applications such as airline reservations, stock control, and banking. It can support applications involving large numbers of terminals (thousands) and very large data bases (gigabytes). The CICS General Information manual [3] gives the following description.

> CICS/VS provides (1) most of the standard functions required by application programs for communication with remote and local terminals and subsystems; (2) control for concurrently running user application programs serving many online users; and (3) data base capabilities . . .

CICS is general purpose in the sense that it provides the primitives of transaction processing, and an individual application is implemented by writing a program invoking these primitives. The primitives are similar to operating system calls, but are at a higher level: they also provide such facilities as security checking, logging and error recovery.

CICS has been in use since 1968, and has undergone continuous development during its lifetime. In the original implementation, the application programmer's interface was at the level of control blocks and assembler macro calls. This is referred to as the macro level application programmer's interface. In 1976 a new interface, the command level application programmer's interface, was introduced. It provides a cleaner interface which does not require the application programmer to have knowledge of the control blocks used in the implementation of the system. The command level interface is the subject of our work on specification.

CICS is supported on a number of IBM operating systems: DOS/VSE, MVS, and MVS/XA, in such a way that application programs written using the application programmer's interface may be transferred from one environment to another without recoding. In addition, the command level interface supports a number of programming languages: PL/I, Cobol, Assembler and RPG II. This is achieved by the use of a preprocessor that translates programs containing CICS commands into the appropriate statements in the language being used (usually a call on a CICS module). Hence the application programmer's interface provides a level of abstraction that hides a number of significantly different implementations.

The command level interface is split up into a number of relatively independent modules responsible for controlling various resources of the system. The formal specification work has so far concentrated on specifying individual modules in relative isolation. Of the sixteen modules comprising the command level interface three: temporary storage, exceptional condition handling, and interval control, have been specified. Temporary storage provides facilities for setting up named temporary storage queues that may be used to communicate information between transactions or as temporary storage by a single transaction. Exceptional condition handling provides facilities to handle exceptions raised by calls on CICS commands in a manner similar to PL/I condition handling. Interval control provides facilities to set up time-outs and delays, as well as to start a new transaction at a given time and pass data to it.

With the large number of CICS systems around the world, the usage of the CICS command level application programmer's interface is on a par with many programming languages. As with programming languages, it is important that the interface be clearly specified in a manner independent of a particular implementation.

USES OF FORMAL SPECIFICATION

The work reported in this paper deals with specification of parts of an already existing system. Before considering the benefits of specification when applied to *existing* software we will briefly review the benefits of specification in general. (For a more detailed discussion see [8].)

In software development a formal specification can be used by

- designers - to formulate and experiment with the design of the system;
- implementors - as a precise description of the system being built, particularly if there is more than one implementation;

- documentors - as an unambiguous starting point for user manuals; and
- quality control - for the development of suitable testing strategies.

Using a specification, the designer of a system can reason about properties of the system before development starts; and during development, formal verification that an implementation meets its specification can be carried out.

When an existing system is being specified there are both short and long term benefits. In the short term performing the specification

1. uncovers those parts of the existing manuals that are either incomplete or inconsistent, and
2. gives insights into the anomalies of the existing system and can suggest ways in which the system could be improved.

In the longer term the specification can be used

1. for reimplementation of all or part of the system,
2. as a basis for discussing and developing specifications for changes or additions to the system, and
3. to provide a model of the functional behaviour of the system suitable for educating new staff.

Re-implementation may involve a new machine architecture, programming language, or operating system, or a restructuring to take advantage of multi-processor or distributed systems. As the specification is implementation independent, it provides a suitable starting point for each of the above alternatives.

When changes or additions to the system are to be made, new specifications can be developed with reference to the previous specification. This process will give insights into the effect of the changes and their interaction with existing parts of the system.

As the specification is a formal document it provides a more precise description for communication between the designers than natural language descriptions. This should help to reduce misunderstandings between the people involved.

Experimentation with specification provides a much quicker and cheaper method of investigating a number of alternative changes to the system than implementing the changes. On the other hand, because the specification is implementation independent, it cannot provide direct answers to questions of how difficult the changes will be to implement, or their impact on the performance of the system. However, as it is at a high level of abstraction it can give a better insight into the interaction of changes with other components of the system; it is just these high level interactions which get lost in the detail of implementation.

While working predominantly at a more abstract level the specifier must be experienced in implementation and should be aware of the implementation consequences of his decisions. Those parts of the specification for which the implementation consequences are unclear should be further investigated before detailed implementation is begun.

THE SPECIFICATION PROCESS

The starting point for our specification work was the CICS command level application programmer's reference manual [2]. The style of this manual is a combination of formal notation describing the syntax of commands and informal English explanations of the operation of the commands. We developed our initial specification of a module of the system by reference to the corresponding section of the manual. The main goal was to come up with a mathematical model of the module that is consistent with its description in the manual. This involves forming a crude initial model of the module and extending it to cover operations (or facets of operations) not initially dealt with, or refining or redesigning the specification as inconsistencies are discovered between it and the manual.

In attempting the initial specification, questions arose that were not satisfactorily answered by the manual. At this stage a list of questions was prepared and an expert on that module of the system (along with the source code) was consulted to answer the questions according to the current implementation.

Questions can arise because

1. the manual is incomplete or vague,

2. the manual is not explicit as to whether possible special cases are treated normally or not,

3. the manual is itself inconsistent, or

4. the chosen mathematical model is inconsistent with the manual in some small way; either the model or the manual is incorrect.

As the system has been in use for some time the answers to the more straightforward questions about its operation have already found their way into the manual. Hence most questions that arose in the specification process were rather subtle and required reference to the source code of the module to be satisfactorily answered. Some of the questions led to inconsistencies being discovered between the manual and the implementation. These inconsistencies could either be errors in the manual or bugs in the implementation. Which way they should be classified depends on the original intent of the designer.

The specification was also given to people experienced in formal specification who gave comments on its internal consistency, style, and they suggested ways in which the specification could be simplified or improved. They were also given a copy of the relevant section of the manual to read *after* they have understood the specification, and were asked to point out any inconsistencies they discovered between it and the specification.

The answers to questions and the review of the specification led to revision of the specification which led to further questions and further review and so on.

Notation

The specification language Z [1, 6, 7] developed in the Programming Research Group at Oxford University is the primary notation that has been used in this specification work. The formal basis of the notation is elementary set theory. People familar with set theory from mathematics should have little trouble understanding the specifications. A summary of the notations used is given in appendix 2.

The style of the specification document is a mixture of formal Z and informal explanatory English. The formal parts of the specification, given in Z, are surrounded in the text by boxes so that they stand apart from the explanatory surrounds and may be more easily found for reference purposes. To make a readable specification, both formal and informal parts are necessary; the formal text can be too terse for easy reading and often its purpose needs to be explained, while the informal natural language explanation can more easily be vague or ambiguous and needs the precision of a formal language to make the intent clear. The informal text provides the link between formality and reality without which the formal text would just be a piece of mathematics. To create a good specification the structuring of the specification and the composition and style of the informal prose are as important as the formal text.

The aim is to provide a specification at a high level of abstraction and thus avoid implementation details. The specification should reveal the operation of the system a small portion at a time. These portions can be progressively combined to give a specification of the whole. This style of presentation is preferred to giving a monolithic specification and trying to explain it; the latter can be rather overwhelming and incomprehensible since there are too many different facets to understand ar once. It is hoped that by giving the specification in small portions each piece can be understood and when the pieces are put together the understanding of the parts that has already been gained can lead more easily to an understanding of the whole.

For more complex specifications that are developed via numerous small steps understanding the whole can be quite difficult, as one needs to remember the function of all the parts and understand the way in which they are combined. In such cases it can be useful to provide both a portion by portion development of the specification and an expanded monolithic specification as well. The latter is more assailable after one has been through a piece by piece development and has an understanding of its various components.

A SAMPLE SPECIFICATION

As a sample of the type of specification produced we will look in detail at the specification of exceptional condition handling within CICS. The exception check mechanisms of CICS are similar to those provided by PL/I [4]. This module was chosen for exposition because it is one of the smaller modules in the system. The manual entry on which the specification was initially based is given in appendix 1 and the notation used in this example is described in appendix 2. The specification given here is a final product of a specification process described in the previous section.

Exceptional Conditions Specification

Exceptional conditions may arise during the execution of a CICS command. A transaction may either set up an action to be taken on a condition by using a `Handle Condition` command, or it may specify that the condition is to be ignored by using an `Ignore Condition` command. If a condition has been neither handled nor ignored, then the default action for that condition will be used.

For example, to handle condition x with action y we can use

```
Handle Condition(c=x, a=y)
```

where the keyword parameter "c=" gives the condition and "a=" gives the action. To ignore condition z we use

```
Ignore Condition(c=z)
```

We introduce the set CONDITION, which contains all the exceptional conditions that may occur, and also contains two special conditions: *success* - the condition that indicates that a command completed normally, and error - a catchall condition that might be used if the exceptional condition that occurred is not handled.

We also introduce the set ACTION which contains all actions that could be taken in response to some exceptional condition. The exact nature of ACTION will not be discussed in detail here. For each programming language supported by CICS it has a slightly different meaning, but for all the languages an action is represented by a label which is given control. There are five special actions used in this specification: *nil* - indicating a normal return (i.e., no action), *abort* - the action that abnormally terminates a transaction, *wait* - indicating that the transaction is to wait until the operation can be completed normally (e.g., wait until space becomes available), and *system* - used to simplify the specification of the Handle Condition command.

The State

The state of the exception controlling system can be defined by

```
┌─ Exceptions ──────────────────────────────
│ Handler : CONDITION ⇸ ACTION
├──────────────
│ Handler(success) = nil
└───────────────────────────────────────────
```

The mapping Handler gives the action to be taken for those conditions that have been set up by either an Ignore Condition or Handle Condition command. The handling action for condition *success* is always *nil* (i.e., return normally). The action for other conditions is determined by some fixed function

Default : CONDITION → ACTION

We state two properties of Default:

Default(*error*) = *abort*
rng(Default) = { *nil*, *abort*, *wait* }

The default action for the special condition error is to abort and the only default actions are *nil*, *abort*, and *wait*.

The initial state of the exception handling system for a transaction is given by

```
┌─ Initial ─────────────────────────────────
│ Exceptions
├──────────────
│ Handler = { success ↦ nil }
└───────────────────────────────────────────
```

The initial state of the handler is to return normally if the operation completes successfully.

As an example, if starting in the initial state the commands

```
Handle Condition(c=x, e=y)
Ignore Condition(c=z)
```

are executed, then the final state will satisfy

$$\text{Handler} = \{\ x \mapsto y,\ z \mapsto \textit{nil},\ \textit{success} \mapsto \textit{nil}\ \}$$

The Handle Condition sets up a mapping from condition x to action y and the Ignore Condition maps condition z onto the *nil* action.

The Operations

The two operations, Handle Condition and Ignore Condition, work directly on the above state. We describe a state change using the following schema, which is called "ΔExceptions" (Δ for change).

```
── ΔExceptions ──────────────────────────
 Exceptions
 Exceptions'
─────────────────────────────────────────
```

Exceptions represents the state of the exception handling system before an operation and Exceptions' the state after. (Appendix 2 contains an expansion of the above schema.)

The operation Handle Condition is used to set up the action, e, to be performed on a particular exceptional condition, c; it is defined as

```
── Handle Condition ──────────────────────────────────
 ΔExceptions
 c : CONDITION
 a : ACTION
──────────────
 c ≠ success ∧ e ∉ { nil, abort, wait } ∧
 (a=system) → Handler' = Handler ⊕ { c ↦ Default(c) },
              Handler' = Handler ⊕ { c ↦ a }
──────────────────────────────────────────────────────
```

The first predicate gives the pre-condition for the operation: the special condition *success* cannot be handled, and the special actions *nil*, *abort*, and *wait* cannot be given as handling actions. The second predicate describes the effect of the operation: if the action to be set up is specified as *system*, then instead the default action for the given condition will be set up as the handler for that condition; otherwise the supplied action, a, will be set up. For example, if the following command is executed in the initial state

$$\mathsf{Handle\ Condition}(\mathsf{c}=\mathsf{x},\ \mathsf{a}=\mathit{system})$$

where $\mathsf{Default}(\mathsf{x}) = \mathit{wait}$, the resulting state will satisfy

$$\mathsf{Handler} = \{\ \mathsf{x} \mapsto \mathit{wait},\ \ \mathit{success} \mapsto \mathit{nil}\ \}$$

The operation to specify that an exceptional condition is to be ignored is given by

```
┌─ Ignore Condition ──────────────────────────────
│ ΔExceptions
│ c : CONDITION
├─────────────────
│ c ≠ success
│ Handler′ = Handler ⊕ { c ↦ nil }
└─────────────────────────────────────────────────
```

The special condition *success* cannot be specified in an Ignore Condition command. The action to be taken on an ignored condition is to return normally (i.e., *nil*).

Exception Checking

Exception handling can take place on any CICS command except Handle Condition and Ignore Condition themselves. We need to describe the exception checking that takes place on all other commands. The exception checking process determines the action, a, to be taken on completion of a command. The value of a is dependent on the condition, c, returned by the command, and the current state of the exception handling mechanism. In addition, any command may specify whether all exceptions are to handled or not for just the execution of that command. In describing the checking process we will include the Boolean variable handle to indicate this. The following defines the (complex) exception checking mechanism which is included in the definition of each operation (other than Handle Condition and Ignore Condition).

```
┌─ Exception Check ──────────────────────────────────────┐
│  Exceptions
│  handle : Boolean
│  c : CONDITION
│  a : ACTION
├──────────────
│  ¬handle              → a = nil,
│  c ∈ dom Handler      → a = Handler(c),
│  Default(c) ≠ abort   → a = Default(c),
│  error ∈ dom Handler  → a = Handler(error),
│                         a = abort
└────────────────────────────────────────────────────────┘
```

If exceptions are not being handled for the command (¬handle) the action is to return normally; otherwise the action is determined from the exception handler. If the condition, c, has been ignored or handled (including the case where the handle action was specified as *system*) then the corresponding handler action is used. Otherwise, if the default action for the condition is not *abort* the default is used, else if the special condition error is handled its handler action is used, otherwise the action will be *abort*.

QUESTIONS RAISED DURING SPECIFICATION

The questions raised about the system during the specification process are an important benefit of the process. They indicate problems either in the documentation of the system or in its logical design, and provide the people responsible for maintaining the system with immediate feedback on problem areas.

In writing a formal specification one is creating a mathematical model of what is being specified, and in creating such a model one is encouraged to be more precise than if one were writing in a natural language. Because of the required precision, questions are raised during the specification process that are not answered by reference to the less formal manual. In fact, the task of formal specification is demanding enough to raise most of the questions about the functional behaviour of the system that would be raised by an attempt to implement it. The effort required for a specification, however, is considerably less than that required for an implementation.

We will now discuss some of the questions that were raised during the specification work on CICS modules. It is interesting to note that most of the questions raised required the expert on the module to refer to the source code to give a conclusive answer. We will begin with the questions about exceptional conditions, then a question about interval control, and finally a question about the interaction between temporary storage and exceptional conditions.

Exceptional Conditions

We will first list some questions that were raised during the specification of exceptional condition handling and then examine one of the more interesting questions in detail. All these questions were resolved in producing the specification given in the previous section.

1. What is the range of possible default actions?

2. Is the default action for a particular condition the same for all commands that can raise that condition?

3. Can the special condition *error* be ignored?

4. Is the action for condition *error* only used if the default system action on a condition is *abort* ?

5. If executed from the initial state does the sequence

```
Handle Condition(c=x, a=y)
...
Handle Condition(c=x, a=system)
```

   return the handler to the initial state?

The reader is invited to try to answer these questions from the manual entry given in appendix 1 and then from the specification given in the previous section. We will now look in detail at question 5 above. It shows a subtle operation of the exceptional conditions mechanism that is counter-intuitive.

In an earlier model of the Handle Condition command the following incorrect line was used in the specification.

$$(a = \mathit{system}) \rightarrow \mathsf{Handler}' = \{c\} \mathbin{⩤} \mathsf{Handler}$$

That is, if the action is *system* then the entry for the condition c is removed from the handler ($c \notin \mathrm{dom}\ \mathsf{Handler}'$). The final model contains the line

$$(a = \mathit{system}) \rightarrow \mathsf{Handler}' = \mathsf{Handler} \oplus \{\ c \mapsto \mathsf{Default}(c)\ \}$$

In this version, if the action is *system* the entry in the handler for condition c is set up to be Default(c) (therefore $c \in \mathrm{dom}\ \mathsf{Handler}'$).

To see the effect of the difference we need to look at the Exception Check mechanism given in the previous section. If we use the second line above then the action when that exception c occurs will be Default(c) (assuming handle is true). In the earlier model, however, the action also depends on whether a handler has been set up for the special condition *error*: the action will be Default(c) unless Default(c) is *abort* and *error* $\in$ dom Handler, in which case the action will be Handler(error). The difference between the two versions is subtle and the reader is encouraged to study the definitions of Handle Condition and Exception Check in order to understand the difference.

The exception check mechanism is quite complex. None of the people experienced with CICS who were questioned about exceptional condition handling were aware of the problem detailed above. It is interesting to conjecture why this is so. The most plausible explanation is that the operation of the exception check mechanism is counter-intuitive. For example, the sequence given in question 5:

```
Handle Condition(c=x, a=y)
...
Handle Condition(c=x, a=system)
```

does not leave the exceptional condition handler in its initial state if the default action for condition x is *abort* and a handler has been set up for the special condition *error*; before the above sequence the *error* handler will be used on an occurrence of condition x, but after, the action Default(x) (i.e., *abort*) will be used on an occurrence of x.

If the above sequence did restore the exception condition handler to its initial state, then it could be used to temporarily handle condition x for the duration of the statements between the Handle Condition commands. This form of operation is more what people using the exceptional conditions module expect.

The Exception Check mechanism is so complex that most readers of either the manual or the specification given in the previous section do not pick up the above subtle operation unless it is explicitly pointed out in some form of warning. This is probably a good argument in favour of revising exception handling to be more intuitive.

The discussion about question 5 above also raises the point that a specification can be incorrect. This case shows one advantage of getting a second opinion on the specification and how it compares to the manual, from a person experienced in formal specification. It is significant that the reviewer reads the specification before reading the manual. The reviewer's mental model of the system is thus based on the mathematical model in the specification. When the reviewer reads the manual looking for inconsistencies with the specification, any questions that arise can be answered by consulting the precise model given in the specification. This contrasts with the person writing the specification who forms a model from the manual and often has to consult other sources to answer questions that arise. Getting a second opinion on the specification and how it compares to the manual is an important ingredient for increasing confidence in the accuracy and readability of the specification.

Interval Control

As another example we will consider one of the problems raised during the specification of the CICS interval control module. Interval control is responsible for operations that deal with the interval timer. The operations provided by interval control can be split logically into two groups: those to do with starting new transactions at specified times, and those to do with time-outs and delays.

In specifying a module of the system we define the state components of the module (in the case of exceptional conditions there was only one state component, Handler). The state components of interval control can be split into two groups that are concerned respectively with the two groups of interval control operations. For the most part, operations only refer to or change components of the corresponding state. One exception is the command Start (to start a new transaction) which in some circumstances changes the time-out state components. This can be considered to be a carefully documented anomaly of the current implementation. Both the implementation and documentation could be simplified if the `Start` command did not destroy the current time-out. More importantly, removal of this interaction would lead to a more useful time-out mechanism, as time-outs would not be affected by a transaction start.

This anomaly is interesting as it points out an unwanted interaction between different parts of a module. In attempting to write the specification this interaction stood out because it involved the `Start` operation using the time-out state. This form of interaction between parts of modules tends to be pinpointed in the formal specification process as the offending operations require access to state information other than that of the part to which they belong.

Two further facts reinforce the view that the current operation of the `Start` command is not the most desirable: if the new transaction is to be started on a different computer system to the one issuing the `Start` command, or if the start is protected (from the point of view of recovery on system failure) then the start does not destroy the current time-out. Ideally we do not want to have to specify distributed system and recovery effects individually with each operation. We would like to add extra levels of abstraction to describe these effects for the whole system.

Interaction Between Modules

As an example of an interaction between two CICS modules we will consider an interaction between exceptional conditions and temporary storage. When temporary storage runs out of space it can raise the exceptional condition *nospace*. This will be processed in the normal way if it has been explicitly handled; the default action, however, is to wait until space becomes available.

Thus the specification of the temporary storage operations that can lead to a *nospace* exception require access to the exceptional conditions state to determine whether or not the *nospace* exception is handled; if it is handled it can occur, but if it is not, it cannot. These operations would more simply be specified (and implemented) if they had an extra parameter indicating whether or not to wait. It is interesting to note that in the implementation such temporary storage commands are transformed into a call with an additional parameter after the exception handling state has been consulted. It is also interesting that these commands were not correctly implemented if the *nospace* exception were ignored.

Interactions between modules of the system are pinpointed during the formal specification process (just as they would be in an implementation) as an operation from one module will need access to the state components of another. Any such interactions discovered during the specification process should be examined closely as they indicate a breakdown in the modular structure of the system.

PROBLEMS WITH SPECIFICATION

This section will examine the problems encountered in applying the formal specification techniques themselves - in contrast to the previous section, which examined problems with the system being specified. The problems encountered in applying specification techniques can be split into communication problems between the people involved, the general problem of achieving the "right" level of abstraction in the specification, and more technical problems related to the particular specification technique.

Communication Problems

As specifiers from a university working with a commercial development laboratory we faced a communications problem. Both parties have their own language: the specifiers use the language of mathematics based on set theory, while the developers use terminology and concepts specific to the system which they are developing. The communication problem is in both directions. This requires education of each party in the language of the other.

In performing a formal specification the specifier needs to understand what is being specified in order to be able to develop a mathematical model of it. To understand the system he needs to read manuals and consult experts, both of which use IBM and CICS terminology. Once a specification is written the specifier would like to get feedback on its suitability from these same experts. This requires that they need to be educated in mathematics to a level at which they can understand a specification. At the current stage of the project the education has been more in the direction of the specifiers learning about the system. In performing a specification of part of a system the specifier of necessity becomes an expert on the functional behaviour of that part (but not on the implementation of the part).

The "right" level of abstraction

In this context "right" means that a piece of specification conveys the primary function of the part of the system it specifies and is not unduly cluttered with details. Most importantly a specification should not be biased towards a particular implementation. However, getting the right specification also involves choosing the most appropriate model and structuring the specification so that the minute details of the component do not hide the primary function.

We can use hierarchical structuring to achieve this. Details of some facet of a component can be specified separately and then that specification can be referenced by the higher level specification. Different cases of an operation (e.g. the normal case and the erroneous case) can be specified independently and combined to give a specification of the whole.

The structure of a good specification may not correspond to the structure one may use to provide an efficient implementation. In specification one is trying to provide a clear logical separation of concerns, while in implementation one may take advantage of the relationships between logically separate parts to provide an efficient implementation of the combined entity. The intellectual ability required of a good specifier is roughly equivalent to that of a good programmer; however, the view taken of the system must be different.

Technical Problems

The technical specification problems discovered in applying formal specification techniques to CICS in particular were

1. putting together the module specifications to provide a specification of the system as a whole,

2. specifying parallelism,

3. specifying recovery on system failures, and

4. specifying distributed systems.

We shall briefly discuss these in turn.

Putting modules together: Currently, three modules out of the sixteen modules that form the application programmer's interface have been specified and we now feel we have enough insight into the system to consider the problem of putting the specifications of modules together. Each module has state components and a set of operations that work on those state components. Putting the modules together is essentially combining the states together to form the state of the system, and extending the operations of the modules to operations on the whole system.

The problems encountered in putting modules together were

1. avoiding name clashes when the modules are combined,
2. specifying the effect on the whole system state of an operation defined within a module of the system, and
3. coping with situations in which an operation of one module refers to state components of another module.

Parallelism: In our current specifications the operations are assumed to be atomic operations acting on the state of the system. We have a sufficient underlying theory to allow one to reason formally about a single sequential transaction. An area for future research is to extend the theory to allow reasoning about the interactions between parallel processes. The current specifications will still be used but they will need to be augmented with additional specifications which constrain the way in which the parallel processes interact.

Recovery: An important part of a transaction processing system is the mechanism for recovery on failure of the system. The current specifications do not address the problem of recovery. Again we would like to augment the current specifications so that recovery can be incorporated without requiring the existing part of the specification to be rewritten.

Distributed Systems: A number of CICS systems may cooperate to provide services to users. The main facility provided within CICS to achieve this is the ability to execute certain operations or whole transactions on a remote system. While the individual operation specifications could be augmented to reflect remote system execution it was thought better to wait until we had a specification of the system and extend that to a distributed system. To reason effectively about a distributed system we need to be able to reason about parallelism.

CONCLUSIONS

Formal specification techniques have been successfully applied to modules of an existing system and as an immediate benefit have uncovered a number of problems in the current documentation as well as flaws in the current interface design. In the longer term the formal specifications should provide a good starting point for specifying proposed changes to the system, a more precise description for educating new personnel, and a basis for improved documentation.

In part the reason we have been successful in applying our specification techniques was that the modular structure of CICS is quite good, and we have been able to take advantage of this by concentrating on individual modules in relative isolation.

The questions raised during the specification process are the main benefit in the short term of applying formal specification techniques to existing software. They highlight aspects of the system that are incompletely or ambiguously described in the manual, as well as focusing attention on problems with its structure, for example, undesirable interactions between modules.

In the longer term a formal specification provides a precise description which can be used to communicate between people involved with the system. The specification is less prone to misunderstanding than less formal means of communication, such as natural language or diagrams. It can be used as a basis for a new specification incorporating modifications to the original design, and it provides an excellent starting point for people responsible for improving the documentation. (In another group at Oxford work on incorporating formal specifications into user manuals is being done by Roger Gimson and Carroll Morgan [5].)

The time required to specify a module of the system varied from about 4 weeks for Exceptional Conditions to 12 weeks for Interval Control. The time required was related to the size of the module (the number of operations, etc.) and also to the number and severity of problems raised about the behaviour of the module. The size of a module specification (in pages) turned out to be roughly comparable to the size of the manual entry for the module. The specification sizes ranged from 4 pages (handwritten) for Exceptional Conditions to 16 pages for Interval Control.

The difficulties encountered with tbe specification process itself were the language gap between university and industry, and the problem of achieving the right level of abstraction. There were also a number of more technical specification problems that arose in applying the techniques: the problem of putting together module specifications to provide a specification of the system as a whole, specifying parallelism, specifying recovery on system failure, and specifying distributed systems. These problems are areas for further research.

ACKNOWLEDGEMENT

I would like to thank IBM for their permission to publish this paper and reproduce part of one of their manuals as appendix 1. Several members of the IBM Development Laboratory at Hursley, England assisted the author to understand some parts of CICS; of special note are Peter Alderson, Peter Collins and Peter Lupton.

This work has benefited from consultations with Tony Hoare, Cliff Jones, and Rod Burstall. Tim Clement was responsible for the initial specification of temporary storage. Paul Fertig, Roger Gimsom, John Nicholls and Bernard Sufrin gave useful comments on this paper. Finally, I would like to express my gratitude to Carroll Morgan and Ib Holm Sørensen for their help as reviewers of the specifications, and for their instruction in specification techniques.

REFERENCES

[1] J.-R. Abrial, "The specification language Z: Basic library", Programming Research Group, Oxford University, Oxford, England, *Internal Report*, 1982.

[2] "CICS/OS/VS application programmer's reference manual (Command level)", IBM Corp., 1980.

[3] "CICS/VS general information", IBM Corp., 1980.

[4] "OS PL/I checkout and optimising compilers: Language reference manual", IBM Corp., 1976.

[5] C. C. Morgan, "Using mathematics in user manuals", Programming Research Group, Oxford University, Oxford, England, *Distributed Computing Project Technical Report*, 1983.

[6] C. C. Morgan and B. A. Sufrin, "Specification of the Unix filing system", *IEEE Trans. on Software Engineering*, vol. 10, no. 2, pp. 128-142, March 1984.

[7] I. H. Sørensen, "A specification language", in *Program Specification* (Lecture Notes in Computer Science, Vol. 134), Springer-Verlag, pp. 381-401, 1982.

[8] J. Staunstrup, *Program Specification: Proceedings of a Workshop, Aarhus, Denmark, August 1981* (Lecture Notes in Computer Science, Vol. 134), Springer-Verlag, 1982.

APPENDIX 1

CICS/VS Version 1 Release 5
Application Programmer's Reference Manual (Command Level)
Exceptional Conditions

Exceptional conditions may occur during the execution of a CICS/VS command and, unless specified otherwise in the application program by an IGNORE CONDITION or HANDLE CONDITION command or by the NOHANDLE option, a default action for each condition will be taken by CICS/VS. Usually, this default action is to terminate the task abnormally.

However, to prevent abnoraml termination, an exceptional condition can be dealt with in the application program by a HANDLE CONDITION command. The command must include the name of the condition and, optionally, a label to which control is to be passed if the condition occurs. The HANDLE CONDITION command must be executed before the command which may give rise to the associated condition.

The HANDLE CONDITION command for a given condition applies only to the program in which it is specified, remaining active until the associated task is terminated, or until another HANDLE CONDITION command for the same condition is encountered, in which case the new command overrides the previous one.

When control returns to a program from a program at a lower level, the HANDLE CONDITION commands that were active in the higher-level program before control was transferred from it are reactivated, and those in the lower-level program are deactivated.

Some exceptional conditions can occur during the execution of any one of a number of unrelated commands. For example, IOERR can occur during file-control operations, interval-control operations, and others. If the same action is required for all occurrences, a single HANDLE CONDITION IOERR command will suffice.

If different actions are required, HANDLE CONDITION commands specifying different labels, at appropriate points in the program will suffice. The same label can be specified for all caommands, and fields EIBFN and EIBRCODE (in the EIB) can be tested to find out which exceptional condition has occurred and in which command.

The IGNORE CONDITION command specifies that no action is to be taken if an exceptional condition occurs. Execution of a command could result in several conditions being raised. CICS/VS checks these in a predetermined order and only the first one that is not ignored (by an IGNORE CONDITION command) will be passed to the application program.

The NOHANDLE option may be used with any command to specify that no action is to be taken for any condition resulting from the execution of that command. In this way the scope of the IGNORE CONDITION command covers specified conditions for all commands (until a HANDLE CONDITION for the condition is executed) and the scope of the NOHANDLE option covers all conditions for specified commands.

The ERROR Exceptional Condition

Apart from the exceptional conditions associated with individual commands, there is a general exceptional condition named ERROR whose default action also is to terminate the task abnormally. If no HANDLE CONDITION command is active for a condition, but one is active for ERROR, control will be passed to the label specified for ERROR. A HANDLE CONDITION command (with or without a label) for a condition overrides the HANDLE CONDITION ERROR command for that condition.

Commands should not be included in an error routine that may give rise to the same condition that caused the branch to the routine; special care should be taken not to cause a loop on the ERROR condition. A loop can be avoided by including a HANDLE CONDITION ERROR command as the first command in the error routine. The original error action should be reinstated at the end of the error routine by including a second HANDLE CONDITION ERROR command.

Handle Exceptional Conditions (HANDLE CONDITION)

```
HANDLE CONDITION condition [ (label) ]
                 [ condition [ (label) ] ]
                      . . .
```

This command is used to specify the label to which control is to be passed is an exceptional condition occurs. It remains in effect until a subsequent IGNORE

CONDITION command for the condition encountered. No more than 12 conditions are allowed in the same command; additional conditions must be specified in further HANDLE CONDITION commands. The ERROR condition can also be used to specify that other conditions are to cause control to be passed to the same label. If "label" is omitted, the default action for the condition will be taken.

The following example shows the handling of exceptional conditions, such as DUPREC, LENGERR, and so on, that can occur when a WRITE command is used to add a record to a data set. DUPREC is to be handled as a special case; system default action (that is, to terminate the task abnormally) is to be taken for LENGERR; and all other conditions are to be handled by the generalized error routine ERRHANDL.

```
EXEC CICS HANDLE CONDITION
          ERROR(ERRHANDL)
          DUPREC(DUPRIN)
          LENGERR
```

If the generalized error routine can handle all exceptions except IOERR, for which the default action (that is, to terminate the task abnormally) is required, IOERR (without a label) would be added to the above command.

In an assembler-language application program, a branch to a label caused by an exceptional condition will restore the registers in the application program to their values at the point where the EXEC interface program is invoked.

In a PL/I application program, a branch to a label in an inactive procedure or in an inactive begin block, caused by an exceptional condition, will produce unpredictable results.

Handle Condition Command Option

condition [(label)] "condition" specifies the name of the exceptional condition, and "label" specifies the location within the program to be branched to if the condition occurs. If this option is not specified, the default action for the condition is taken, unless the default action is to terminate the task abnormally, in which case the ERROR condition occurs. If the option is specified without a label, any HANDLE CONDITION command for the condition is deactivated, and the default action taken if the condition occurs.

Ignore Exceptional Conditions (IGNORE CONDITION)

```
IGNORE CONDITION  condition
                [ condition ]
                    . . .
```

This command is used to specify that no action is to be taken if an exceptional condition occurs. It remains in effect until a subsequent HANDLE CONDITION command for the condition is encountered. No more than 12 conditions are allowed in the same command; additional conditions must be specified in further IGNORE CONDITION commands. The option "condition" specifies the name of the exceptional condition that is to be ignored.

APPENDIX 2
NOTATION

Schemas

A schema has the general form

```
┌─ Name ──────────────────────────────┐
│   Declarations
├────────────────────
│   Predicates
└─────────────────────────────────────┘
```

where the variable declarations are of the form

```
identifier : type
```

and the predicates give the properties of, and relationships between, the variables.

A schema may be used to describe either a state or an operation. To describe a state the declared variables form the components of the state and the predicates give the invariant properties of the state. For an operation the declarations consist of the initial state components, the final state components, and the inputs and outputs of the operation. As a convention the final state component names are dashed versions of the initial state component names. For an operation the predicate part describes the relation between the inputs, outputs, and initial and final states.

A schema S may be included within another schema T. This has the effect of including all the variables declared in S in the declarations of T and of including all the predicates of S in the predicates of T. A schema name may be decorated (e.g., dashed). This has the effect of decorating in a similar way (e.g., dashing) all the declared variables both in their declaration and their uses within the predicates. For example, the schema ΔExceptions given in the sample specification in the body of the paper is equivalent to

```
┌─ ΔExceptions ───────────────────────┐
│   Handler  : CONDITION ⇸ ACTION
│   Handler' : CONDITION ⇸ ACTION
├────────────────────
│   Handler(success)  = nil
│   Handler'(success) = nil
└─────────────────────────────────────┘
```

Logic

Within the predicate part we may use the operators

$\wedge$	- and
$\vee$	- or
$\neg$	- negation
$\Rightarrow$	- implication
$=$	- equality
$\neq$	- inequality
$\forall x{:}T \bullet P$	- for all x of type T, P holds
$\exists x{:}T \bullet P$	- there exists an x of type T such that P holds
$c \rightarrow x, y$	- conditional expression

For the conditional expression if c is true the value of the conditional expression is x; otherwise it is y.

Sets

We may construct a set by listing its elements within braces:

$\{ x, y, z \}$

or by giving some property that only elements of the set have

$\{ x{:}T \mid P(x) \}$.

We may test the following

$\in$	- membership, e.g., $1 \in \{1,2,3\}$
$\notin$	- non-membership, e.g., $2 \notin \{1,3,5\}$
$\subseteq$	- subset, e.g., $\{2,3\} \subseteq \{2,3,4\}$

and perform the following operations on sets, given A and B subsets of T

$\cup$	- set union: $A \cup B = \{x{:}T \mid x \in A \vee x \in B\}$
$\cap$	- set intersection: $A \cap B = \{x{:}T \mid x \in A \wedge x \in B\}$
$-$	- set difference: $A - B = \{x{:}T \mid x \in A \wedge x \notin B\}$

Functions

We may declare a function f from a set A to a set B by

$$f : A \rightarrow B$$

For each element $x \in A$, $f(x)$ is the value of the function f at x ($f(x) \in B$).

If a function f is not defined for all elements of A (i.e., f is a partial function) then we write

$$f : A \twoheadrightarrow B$$

The domain of definition of f

$$\text{dom } f$$

is that subset of A ($\text{dom } f \subseteq A$) for which the function f is defined.

The range of f is that subset of B ($\text{rng } f \subseteq B$) containing exactly those values $b \in B$ such that there exists an $x \in \text{dom } f$ such that $f(x) = b$. That is

$$\text{rng } f = \{ b \in B \mid (\exists x : A \cdot f(x) = b) \}$$

The notation

$$\{ x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n \}$$

where each x_k is distinct, defines a function whose domain of definition is the set of x_k's:

$$\text{dom } \{ x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n \} = \{ x_1, x_2, \ldots, x_n \}$$

and the value of the function at x_k is y_k:

$$\{ x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n \}(x_k) = y_k.$$

The notation

$$f \oplus g$$

stands for function f overridden by function g (we assume functions f and g are of the same type). The function $f \oplus g$ is defined at a point if either f or g is defined at that point:

$$\text{dom } (f \oplus g) = \text{dom } f \cup \text{dom } g$$

If g is defined at x then the value of $f \oplus g$ is $g(x)$; otherwise, if f is defined at x the value of $f \oplus g$ is $f(x)$:

$$\begin{aligned} &x \in \text{dom } g && \Rightarrow (f \oplus g)(x) = g(x) \\ &x \notin \text{dom } g \wedge x \in \text{dom } f && \Rightarrow (f \oplus g)(x) = f(x) \end{aligned}$$

The notation

$$s \ntriangleleft f$$

stands for the function f with all elements of its domain that are in the set s removed

$$\begin{aligned} &\text{dom } (s \ntriangleleft f) = (\text{dom } f) - s \\ &x \in \text{dom } (s \ntriangleleft f) \Rightarrow (s \ntriangleleft f)(x) = f(x) \end{aligned}$$

CICS TEMPORARY STORAGE

Abstract

Temporary storage provides facilities for storage of information in named "queues". The operations that can be performed on an individual queue are either the standard queue-like operations (append to the end and remove from the beginning), or array-like random access read and write operations.

A Single Queue

An element of a queue is a sequence of bytes.

$$TSElem \mathrel{\widehat{=}} \mathrm{seq}\ Byte$$

A single queue may be defined by

$$\begin{array}{|l}
\multicolumn{1}{l}{TSQ} \\
\hline
ar \ : \ \mathrm{seq}\ TSElem \\
p \ \ : \ \mathbb{N} \\
\hline
p \leq |ar| \\
\hline
\end{array}$$

The array ar contains the items in the queue. The size of the array is always equal to the number of append operations that have been performed on the queue since its creation - independently of the number of other (remove, read, or write) operations. The pointer p keeps track of the position of the item which was last removed or read from the queue.

The initial state of a queue is given by an empty array and a zero pointer.

$$TSQ\_Initial \mathrel{\widehat{=}} [\ TSQ \mid (ar = [\,]) \wedge (p = 0)\ ]$$

Operations

We will define four operations on a single TSQ. The definitions of these operations will use the schema

$$\Delta TSQ \mathrel{\widehat{=}} TSQ \wedge TSQ'$$

ΔTSQ (Δ for change) defines a before state TSQ, with components ar and p (satisfying $p \leq |ar|$), and an after state TSQ′, with components ar′ and p′ (satisfying $p' \leq |ar'|$). The definitions of the operations follow.

```
┌─ Append_0 ──────────────────────────
│ ΔTSQ
│ from? : TSElem
│ item! : ℤ
├─────────────────
│ ar' = ar ⁀ [from?] ∧
│ item! = |ar'| ∧
│ p' = p
└─────────────────────────────────────
```

The new element from? (a "?" *at the end of a name indicates an input*) is appended to the end of ar to give the new value of the array. The position of the new item is returned in item! (a "!" *at the end of a name indicates an output*). The pointer position is unchanged.

```
┌─ Remove_0 ──────────────────────────
│ ΔTSQ
│ into! : TSElem
├─────────────────
│ p < |ar| ∧
│ p' = p + 1 ∧
│ into! = ar(p') ∧
│ ar' = ar
└─────────────────────────────────────
```

The pointer must not have already reached the end of the array. The pointer is incremented to the next item in the queue and the value of that item is returned in into!. The contents of the array is *unchanged*.

```
┌ Write₀ ──────────────────────────────
│ ΔTSQ
│ item? : ℤ
│ from? : TSElem
├──────────────
│ item? ∈ 1..|ar| ∧
│ ar′ = ar ⊕ { item? ↦ from? } ∧
│ p′ = p
└──────────────────────────────────────
```

The position item? must lie within the bounds of the current array. The item at that position in ar is overridden by the value of from? to give the new value of the array. The pointer position is unchanged.

```
┌ Read₀ ───────────────────────────────
│ ΔTSQ
│ item? : ℤ
│ into! : TSElem
├──────────────
│ item? ∈ 1..|ar| ∧
│ into! = ar(item?) ∧
│ p′ = item? ∧
│ ar′ = ar
└──────────────────────────────────────
```

The value of the item at position item?, which must lie within the bounds of the array, is returned in into!. The pointer position is updated to be item?. The array is unchanged.

In the above, all the operations have been specified in terms of the array ar and pointer p. While this is reasonable for the Read and Write operations it does not show the queue-like nature of the Append and Remove operations. Let us uow show that the queue-like operations are the familiar oues. We can define a standard queue by

```
Q ≙ seq TSElem
```

The standard append to the end of a queue operation is given by

```
┌ Standard_Append ──────────────────
│ ΔQ
│ from? : TSElem
├───────────────
│ q′ = q ⁀ [from?]
└───────────────────────────────────
```

where ΔQ ≙ [q, q′ : Q].

The standard remove from the front of the queue operation is given by

```
┌ Standard_Remove ──────────────────
│ ΔQ
│ into! : TSElem
├───────────────
│ q = [into!] ⁀ q′
└───────────────────────────────────
```

The predicate in the above specification may be unconventional to some readers. It states that the value of the queue before the operation is equal to the value returned in into! *catenated with the value of the queue after the operation. This form of specification more closely reflects the symmetry between* Standard_Append *and* Standard_Remove *than the more conventional*

```
into! = head(q)
q′ = tail(q)
```

To see the relatioiship between standard queues and temporary storage queues we need to formulate the correspondence between the respective states.

```
┌ QLike ────────────────────────────
│ q : Q
│ TSQ
├───────────────
│ q = tailᵖ(ar)
└───────────────────────────────────
```

A standard q corresponds to the array ar with the first p elements removed. Given this relationship between states we will now show the relationship between Append_0 and Standard_Append. What we will show is that if we perform an Append_0 with initial state TSQ and final state TSQ′ then the corresponding standard queue states Q and Q′ (as determined by QLike and QLike′ respectively) are related by Standard_Append. This can be formalised by the following theorem.

Theorem: $\mathsf{Append}_0 \wedge \mathsf{QLike} \wedge \mathsf{QLike}' \vdash \mathsf{Standard\_Append}$

Proof:

1. $q, q' : \mathsf{seq}\ \mathsf{TSElem};\ \mathsf{from?} : \mathsf{TSElem}$	from QLike, QLike′ and Append_0
2. $q' = \mathsf{tail}^{p'}(ar')$	from QLike′
3. $\quad = \mathsf{tail}^{p}(ar \frown [\mathsf{from?}])$	from Append_0
4. $\quad = (\mathsf{tail}^{p}(ar)) \frown [\mathsf{from?}]$	as $p \leqslant \lvert ar \rvert$ from TSQ
5. $\quad = q \frown [\mathsf{from?}]$	from QLike
6. Standard_Append	from (1), (5) $\square$

We can now do the same for Remove.

Theorem: $\mathsf{Remove}_0 \wedge \mathsf{QLike} \wedge \mathsf{QLike}' \vdash \mathsf{Standard\_Remove}$

Proof:

1. $q, q' : \mathsf{seq}\ \mathsf{TSElem};\ \mathsf{into!} : \mathsf{TSElem}$	from QLike, QLike′ and Remove_0
2. $p < \lvert ar \rvert$	from Remove_0
3. $q = \mathsf{tail}^{p}\,(ar)$	from QLike
4. $\quad = [ar(p+1)] \frown (\mathsf{tail}^{p+1}\,(ar))$	from (2) and property of tail
5. $\quad = [\mathsf{into!}] \frown (\mathsf{tail}^{p'}\,(ar'))$	from Remove_0
6. $\quad = [\mathsf{into!}] \frown q'$	from QLike′
7. Standard_Remove	from (1), (6) $\square$

Errors

To allow for errors we can introduce a report to indicate success or failure of an operation. If an error occurs we would like the TSQ to remain unchanged. This can be encapsulated by

```
┌ ERROR ─────────────────────────
│ ΔTSQ
│ report! : CONDITION
├──────────────
│ TSQ' = TSQ
└────────────────────────────────
```

where the set CONDITION contains all the error reports plus the report *Success*. In the operations described above there are three errors that can occur: trying to remove an item from a TSQ with no items left to remove, trying to read or write at a position outside the array, and running out of space to store an item.

```
┌ NoneLeft! ─────────────────────
│ ERROR
├──────────────
│ p = |ar| ∧
│ report! = ItemErr
└────────────────────────────────
```

```
┌ OutofBounds! ──────────────────
│ ERROR
│ item? : ℤ
├──────────────
│ item? ∉ 1..|ar| ∧
│ report! = ItemErr
└────────────────────────────────
```

```
┌ NoSpace! ──────────────────────
│ ERROR
├──────────────
│ report! = NoSpace
└────────────────────────────────
```

If the operations work correctly the report will indicate *Success*.

$$\text{Successful} \mathrel{\widehat{=}} [\ \text{report!} : \text{CONDITION} \mid \text{report!} = \mathit{Success}\ ]$$

The operations given previously can now be combined with the erroneous situations. We will redefine the operations in terms of their previous definitions.

$$\begin{aligned}
\text{Append} &\mathrel{\widehat{=}} (\text{Append}_0 \wedge \text{Successful}) \vee \text{NoSpace!} \\
\text{Remove} &\mathrel{\widehat{=}} (\text{Remove}_0 \wedge \text{Successful}) \vee \text{NoneLeft!} \\
\text{Write} &\mathrel{\widehat{=}} (\text{Write}_0 \wedge \text{Successful}) \vee \text{OutofBounds!} \vee \text{NoSpace!} \\
\text{Read} &\mathrel{\widehat{=}} (\text{Read}_0 \wedge \text{Successful}) \vee \text{OutofBounds!}
\end{aligned}$$

Note that NoSpace! does not specify under what conditions it occurs. The specifications of Append and Write do not allow us to determine whether or not the operation will be successful from the initial state and inpnts to an operation. This is au example of a non-deterministic specification. It is left to the implementor to determine when a NoSpace! report will be returned (we hope it will not be on every call).

Named Queues

We now want to specify a system with more than one queue. A particular TSQ can be specified by name and the above operations performed on it. We will use a mapping from queue names (TSQName) to queues. The state of our system of queues is given by

$$\text{TS} \mathrel{\widehat{=}} \text{TSQName} \mathrel{\rightarrowtail\!\!\!\!\!\rightarrow} \text{TSQ}$$

The initial state of the system of quenes is given by an empty mapping.

$$\text{TS\_Initial} \mathrel{\widehat{=}} \{\}$$

Our operations require updatiug of a particnlar named TSQ. We can introduce a schema, UpdateQ, to encapsulate the common part of npdating for operations on queues that already exist.

```
┌ UpdateQ ──────────────────────────────
│ ΔTS
│ queue? : TSQName
│ ΔTSQ
├──────────────
│ queue? ∈ dom(ts) ∧
│ TSQ = ts(queue?) ∧
│ ts′ = ts ⊕ { queue? ↦ TSQ′ }
└───────────────────────────────────────
```

where $\Delta TS \mathrel{\widehat{=}} [\ ts, ts' : TS\ ]$. Note that UpdateQ specifies that the named queue (alone) is updated but does not specify in what way it is updated. This is achieved by combining it with the single queue operations to get the operation on named queues.

In adding named queues we have added the possibility of a new error: trying to perform operations on non-existent queues. This error is given by

```
┌ NonExistent! ─────────────────────────
│ ΔTS
│ queue? : TSQName
│ report! : CONDITION
├──────────────
│ queue? ∉ dom(ts) ∧
│ ts′ = ts ∧
│ report! = QIdErr
└───────────────────────────────────────
```

Our operations, except AppendQ which is allowed on a non-existent queue, can now be redefined in terms of our previous definitions.

```
RemoveQ ≙  (UpdateQ ∧ Remove) \ ΔTSQ
             ∨  NonExistent!

WriteQ  ≙  (UpdateQ ∧ Write) \ ΔTSQ
             ∨  NonExistent!

ReadQ   ≙  (UpdateQ ∧ Read) \ ΔTSQ
             ∨  NonExistent!
```

The temporary variables in ΔTSQ (ar, p, ar′, p′) are hidden in the signatures of the final operations and the operations inherit the errors from the equivalent single queue operations.

A queue is created by performing an AppendQ operation on a queue that does not exist. The following schema describes the creation of a queue.

```
┌─ CreateQ ──────────────────────────────
│ ΔTS
│ queue? : TSQName
│ TSQ_Initial
│ TSQ'
├─────────────────
│ queue? ∉ dom(ts) ∧
│ ts' = ts ∪ { queue? ↦ TSQ' }
└────────────────────────────────────────
```

Again the relationship between TSQ_Initial (ar, p) and TSQ' (ar', p') is not defined within this schema. This is supplied by Append in the following definition

$$\text{AppendQ} \ \widehat{=}\ ((\text{UpdateQ} \lor \text{CreateQ}) \land \text{Append}) \setminus \Delta\text{TSQ}$$

Note that for a non-existent queue, if an error occurs (i.e. a NoSpace condition), then an empty queue will be created.

In addition to these promoted operations on named queues we have an operation to delete a named queue.

```
┌─ DeleteQ₀ ─────────────────────────────
│ ΔTS
│ queue? : TSQName
│ report! : CONDITION
├─────────────────
│ queue? ∈ dom(ts) ∧
│ ts' = { queue? } ⩤ ts ∧
│ report! = Success
└────────────────────────────────────────
```

An exception occurs if the queue to be deleted does not exist; DeleteQ becomes

$$\text{DaleteQ} \ \widehat{=}\ \text{DeleteQ}_0 \lor \text{NonExistent!}$$

A Network of Systems

Temporary storage queues may be located on more than one system. Let us call the set of all possible system identifiers $SysId$. We can represent temporary storage queues on a network of systems by

$$NTS \mathrel{\widehat=} SysId \twoheadrightarrow TS$$

For a network

$$nts : NTS$$

$dom(nts)$ is the set of systems that share temporary storage queues and for a system with identity $sysid$ such that $sysid \in dom(nts)$, $nts(sysid)$ is the temporary storage state of that system. The operations on temporary storage queues may be promoted to operate for a network of systems by the following schema.

$$\begin{array}{|l}
\text{Network} \\
\hline
\Delta NTS \\
sysid? : SysId \\
\Delta TS \\
\hline
sysid? \in dom(nts) \land \\
ts = nts(sysid?) \land \\
nts' = nts \oplus \{\ sysid? \mapsto ts'\ \} \\
\hline
\end{array}$$

where $\Delta NTS \mathrel{\widehat=} [\ nts, nts' : NTS\ ]$. As with promoting the operations to work on named queues the above schema only specifies which system is updated but not how it is updated. This will be supplied when this schema is combined with the definitions of the operations on a single system.

Network operation also introduces the possibility of an error if the given system does not exist.

```
─ NoSystem! ─────────────────────────────
  ΔNTS
  sysid? : SysId
  report! : CONDITION
─────────────────────
  sysid? ∉ dom(nts) ∧
  nts′ = nts ∧
  report! = SysIdErr
─────────────────────────────────────────
```

The operations on a multiple system are given by

$$
\begin{aligned}
\mathrm{AppendQN}_0 \ &\widehat{=}\ (\mathrm{AppendQ} \wedge \mathrm{Network}) \setminus \Delta\mathrm{TS} \\
&\quad \vee\ \mathrm{NoSystem!} \\[1ex]
\mathrm{RemoveQN}_0 \ &\widehat{=}\ (\mathrm{RemoveQ} \wedge \mathrm{Network}) \setminus \Delta\mathrm{TS} \\
&\quad \vee\ \mathrm{NoSystem!} \\[1ex]
\mathrm{ReadQN}_0 \ &\widehat{=}\ (\mathrm{ReadQ} \wedge \mathrm{Network}) \setminus \Delta\mathrm{TS} \\
&\quad \vee\ \mathrm{NoSystem!} \\[1ex]
\mathrm{WriteQN}_0 \ &\widehat{=}\ (\mathrm{WriteQ} \wedge \mathrm{Network}) \setminus \Delta\mathrm{TS} \\
&\quad \vee\ \mathrm{NoSystem!}
\end{aligned}
$$

The sysid? and queue? name supplied as inputs are not necessarily the ones on which an operation takes place. A queue name on a given system may be marked as actually being located on another (remote) system, possibly with a different name on that remote system. We will model this by the following function which takes the input pair of sysid? aud queue? name and gives the corresponding actual sysid! and queue! name on which the operation will be performed.

$$\mathrm{remote} : (\mathrm{SysId} \times \mathrm{TSQName}) \rightarrow (\mathrm{SysId} \times \mathrm{TSQName})$$

In many cases the input sysid? and queue? name are the actual system and queue name; in these cases remote will behave as the identity.

We will use the following schema to incorporate remote into the operations.

```
┌─ TSRemote ──────────────────────────────
│ sysid?, sysid! : SysId
│ queue?, queue! : TSQName
├──────────────
│ (sysid!, queue!) = remote(sysid?, queue?)
└─────────────────────────────────────────
```

The outputs, sysid! and queue!, of TSRemote form the inputs to the operations. If a sysid? parameter is supplied then the operations on temporary storage queues are defined by

$$\mathrm{AppendQN}_1 \mathrel{\widehat=} \mathrm{TSRemote} \gg \mathrm{AppendQN}_0$$

$$\mathrm{RemoveQN}_1 \mathrel{\widehat=} \mathrm{TSRemote} \gg \mathrm{RemoveQN}_0$$

$$\mathrm{ReadQN}_1 \mathrel{\widehat=} \mathrm{TSRemote} \gg \mathrm{ReadQN}_0$$

$$\mathrm{WriteQN}_1 \mathrel{\widehat=} \mathrm{TSRemote} \gg \mathrm{WriteQN}_0$$

If no sysid? parameter is given then the operations are given by

$$\mathrm{AppendQN}_2 \mathrel{\widehat=} \mathrm{AppendQN}_1[\mathrm{cursysid?}/\mathrm{sysid?}]$$

$$\mathrm{RemoveQN}_2 \mathrel{\widehat=} \mathrm{RemoveQN}_1[\mathrm{cursysid?}/\mathrm{sysid?}]$$

$$\mathrm{ReadQN}_2 \mathrel{\widehat=} \mathrm{ReadQN}_1[\mathrm{cursysid?}/\mathrm{sysid?}]$$

$$\mathrm{WriteQN}_2 \mathrel{\widehat=} \mathrm{WriteQN}_1[\mathrm{cursysid?}/\mathrm{sysid?}]$$

That is, the sysid? parameter is replaced by a parameter giving the identity of the current system (the system on which the operation was initiated).

A note on the current implementation

Each system keeps track of the names of queues that are located on other (remote) systems and for each remote queue the identity of the remote system and the name of the queue on that system. It is possible that the referred request could be for a queue name that is also remote to the referred system, in which case the request will be referred on to yet another system. To find the system on which the queue actually resides we need to follow through a chain of systems until we get to a system on which the queue name is considered local. We can model the implementation by the function

$$\mathsf{rem} : (\mathsf{SysId} \times \mathsf{TSQName}) \nrightarrow (\mathsf{SysId} \times \mathsf{TSQName})$$

which for a sysid and queue name gives the sysid and queue name of the next link in the chain; if a sysid and queue name pair is not in the domain of rem then the chain is finished. The correspondence between rem and remote is given by

$$\mathsf{remote} = \mathsf{repeat\ rem}$$

where repeat applies the function rem repeatedly until the parameter to rem is no longer in the domain of rem

$$\begin{aligned}(\mathsf{repeat}\ f)\ y &= y && \text{if } y \notin \mathrm{dom}(f)\\ &= (\mathsf{repeat}\ f)\ (f\ y) && \text{if } y \in \mathrm{dom}(f)\end{aligned}$$

That is

$$\begin{aligned}\mathsf{remote}(s, q) &= (s, q) && \text{if } (s, q) \notin \mathrm{dom\ rem}\\ &= \mathsf{remote}\ (\mathsf{rem}\ (s, q)) && \text{if } (s, q) \in \mathrm{dom\ rem}\end{aligned}$$

As remote is a total fnnction the equality of remote and (repeat rem) requires that no chain of rem's contains any loop (so that (repeat rem) is also total).

Given the function `rem` if we take the corresponding (curried) function with the following shape

```
r : SysId → (TSQName ⇸ (SysId × TSQName))
```

so that

```
r(s)(q)     = rem(s, q)
dom(r(s)) = { q : TSQName | (s, q) ∈ dom rem }
```

The mapping that needs to be stored on a system `s` is given by `r(s)`, and is of type

```
TSQName ⇸ (SysId × TSQName)
```

Acknowledgements

The work reported in this paper was supported by a grant from IBM. The starting point for this specification was an earlier specification done by Tim Clement. This specification has benefited greatly from the detailed comments of Carroll Morgan and Ib Holm Sørensen.

CICS INTERVAL CONTROL

Abstract

The specification of Interval Control has been split into the specification of a timeout system, and a start/retrieve system, as these are logically different functions.

In specifying the timeout system, events were initially included but it was later discovered that they were logically redundant and should not be part of the Interval Control interface. The version of the timeout system without events is presented here.

There are a number of differences (or omissions) between this specification and the actual interface:

- Remote system aspects are not included.
- A more abstract time parameter is used.
- A more abstract data parameter is used.
- Function management headers were ignored as they are a detail internal to the structure of the data (and not explained anywhere in the manual).
- The "nocheck" and "protect" options to Start have been ignored as they are to do with recovery. It is hoped to upgrade the specification to include recovery aspects at a later stage, again on a CICS-wide basis.
- The "wait" option to Retrieve (and associated "dtimout" mechanism were not specified).

As the specification techniques used here are only suitable for specifying sequential operations, the parts of Interval Control involving concurrent processes are not adequately specified in this document. Interval Control is complicated, as is this specification.

Time

The Interval Control operations are involved with (intervals of) time. We can represent the effect on time of the operations by the following change of time schema.

```
ΔTime________________________________________
| clock, clock' : Time
|_____________________
|
| clock ≤ clock'
|____________________________________________
```

where Time ≙ ℕ. We will assume time is measured in units of, say, seconds. Time cannot decrease.

The operation to determine the current time is given by

```
AskTime______________________________________
| ΔTime
| time! : Time
| report! : Condition
|_____________________
|
| time! = clock ∧
| report! = OK
|____________________________________________
```

Aside: The CICS AskTime operation has no explicit output parameters but rather returns the time of day and the date in the Exec Interface Block (EIB) fields EIBTIME and EIBDATE. A specification should avoid the implementation detail of the EIB and hence an explicit output parameter has been used above. Furthermore, only a single output time! incorporates both the date and time of day information. This is a little more abstract and allows consistent use of time throughout the specification. □

Timeouts

The following version of the timeout system is more abstract than the actual system. This version avoids the need for events to be passed to and fro on operations.

The state required for timeout operations is

```
TO__________________________________
| timeout : Time
| setup,
| cancelled : Boolean
|___________________________________
```

The initial timeout state of a process is given by

$TO_{INIT} \mathrel{\widehat=} [\ TO \mid \neg setup\ ]$

A state change on a timeout operation is given by

$\Delta TO \mathrel{\widehat=} \Delta Time \wedge TO \wedge TO'$

The following operation is used to set up a timeout at `time?`.

```
SetUpTO0____________________________
| ΔTO
| time? : Time
|______________
| clock < time? ∧
| timeout' = time? ∧
| setup' ∧ ¬cancelled'
|___________________________________
```

A timeout can only be setup provided the time has not already passed. The final state records the timeout time and that a timeout has been setup and not yet cancelled.

Aside: The corresponding CICS operation (inappropriately called "Post") has two differences to the above. Firstly, it returns an event, and this specification avoids the need for events. Secondly, the time parameter for the CICS operation may be either relative to the current time or an absolute time in the day (well maybe in the morrow - to quote: "CICS treats as expired a request for an absolute time that is equal to the current time or that precedes the current time by up to six hours. If the specified absolute time precedes the curreut time by more than six hours, CICS adds 24 hours, that is, the requested function is performed at the time specified but on the next day.") In the specification above we have used a time parameter that is consistent with the time used by AskTime and hence it is neither restricted to a 100 hour period nor does it require the complicated definition quoted above. □

To determine if the current timeout has expired we use

```
TestExpiry₀ ─────────────────────────────────────
│ ΞTO
│ hasexpired! : Boolean
├──────────────────
│ setup ∧
│ hasexpired! = (timeout ≤ clock) ∨ cancelled
└─────────────────────────────────────────────────
```

where ΞTO ≙ [ΔTO | TO′ = TO]. In order to test expiry a timeout must have been previously setup. A timeout is considered to have expired either if the time has passed or the timeout has been cancelled.

Aside: This operation is not currently provided explicitly by CICS, rather, to quote: "When the time specified has expired, the timer event control area is posted; that is, its first byte is set to X'40' and its third byte to X'80'." Our TestExpiry operation is an abstraction of a rather low level bit testing operation. Furthermore, the lack of an explicit test operation is a major reason for the introduction of events into the Interval Control interface. If there were a TestExpiry operation then there would be no need for Post to return an event. □

To delay the current process until the previously setup timeout has expired we use

```
Wait₀ ──────────────────────────────────
│ ΔTO
├──────────────────
│ setup ∧
│ ¬setup′ ∧
│ (timeout ≤ clock′) ∨ cancelled′
└────────────────────────────────────────
```

Aside: The equivalent CICS operation Wait Event has an input event parameter "ecaddr". This is not necessary in the above specification as it is assummed we are waiting for the current timeout to expire. In practice, however, the Wait Event operation is also used for process synchronisation. To quote: "This command is used to synchronise a task with the completion of an event initiated by the same task or by another task". However, the manual also states: "No other task should attempt to wait on the event setup by a Post command. The timer event control area can be released for a variety of reasons (e.g. task termination). If this happens, the result of any other task issuing a Wait on the event setup by the Post is unpredictable." In summary, events created by Interval Control are used for synchronisation by some applications but this must be used with great care.

The Wait Event operation can also get an *InvReq* exception "if the specified event control area address is above 16 megabytes for a program executing in 31-bit mode on MVS/XA." By avoiding events we avoid this, but even with events I think we would like to avoid it! □

The operation to delay a process until a given time is

```
Delay$_0$
  ΔTO
  time? : Time
  ────────────
  clock < time? ∧
  (time? ≤ clock') ∨ cancelled'
```

For cancelling timeouts we need two different forms of cancel operation: one, CancelTO, when a process is cancelling its own timeout and the other, CancelTOReq, when a process issues a cancel with a request identifer (see later section for more details) indicating which process' timeout is to be cancelled.

```
CancelTO$_0$                  CancelTOReq$_0$
  ΔTO                           ΔTO
  ────────                      ────────
  setup ∧                       setup ∧
 ¬setup' ∧                      setup' ∧
  cancelled'                    cancelled'
```

In CancelTO$_0$ the timeout does not remain setup while in CancelTOReq$_0$ it does.

Aside: The different operation of Cancel depending on whether there is a request identifier given or not is rather anomalous. It would appear to be a side effect of the current implementation. □

Errors

If the time has already expired on a setup timeout or delay an *Expired* exception can occur.

```
Expired!─────────────────────────────────
│ ΞTO
│ time? : Time
│ report! : Condition
├──────────────
│ time? ≤ clock ∧
│ report! = Expired
└────────────────────────────────────────
```

Aside: There is also an *InvReq* exception for the CICS operations due to an invalid format time parameter. The more abstract time used here avoids such an exception. □

If a TestExpiry, Wait or Cancel operation is performed when a timeout has not been setup we get a *InvReq* exception.

```
NotSetUp!────────────────────────────────
│ ΞTO
│ report! : Condition
├──────────────
│ ¬setup ∧
│ report! = InvReq
└────────────────────────────────────────
```

If the operations do not get an exception then report! will indicate success.

$$Success \mathrel{\widehat{=}} [\ report! : Condition \mid report! = OK\ ]$$

The total timeout system operations are

$$\begin{array}{lcll}
\mathsf{SetUpTO}_1 & \widehat{=} & (\mathsf{Success} \wedge \mathsf{SetUpTO}_0) & \vee\ \mathsf{Expired!} \\
\mathsf{TestExpiry}_1 & \widehat{=} & (\mathsf{Success} \wedge \mathsf{TestExpiry}_0) & \vee\ \mathsf{NotSetUp!} \\
\mathsf{Wait}_1 & \widehat{=} & (\mathsf{Success} \wedge \mathsf{Wait}_0) & \vee\ \mathsf{NotSetUp!} \\
\mathsf{Delay}_1 & \widehat{=} & (\mathsf{Success} \wedge \mathsf{Delay}_0) & \vee\ \mathsf{Expired!} \\
\mathsf{CancelTO}_1 & \widehat{=} & (\mathsf{Success} \wedge \mathsf{CancelTO}_0) & \vee\ \mathsf{NotSetUp!} \\
\mathsf{CancelTOReq}_1 & \widehat{=} & (\mathsf{Success} \wedge \mathsf{CancelTOReq}_0) & \vee\ \mathsf{NotSetUp!}
\end{array}$$

Aside: There is currently no method provided by Interval Control for a process to determine whether a timeout or delay has been cancelled or whether it expired. This could be provided by a *Cancelled* exceptional condition. □

Multiple Processes

In the preceding we have only used the state information of a single process. In order to include request identifiers which allow one process to cancel another's timeout we will extend our state to multiple processes.

$$\mathsf{PTO} \ \widehat{=}\ \mathsf{PId} \twoheadrightarrow \mathsf{TO}$$

where PId is the set of process identifiers. Each process is associated with a unique element of PId. Given a process identifier pid the timeout state associated with the corresponding process is given by pto(pid).

The initial state of the timeout system is given by

$$\mathsf{PTO}_{\mathsf{INIT}} \ \widehat{=}\ \{\}$$

To promote our timeout operations to equivalent ones in the multi-process state acting on a single process within that state we use the following promotion schema.

```
MPTO_________________________________________
 | ΔPTO
 | currentpid? : PId
 | ΔTO
 |____________
 | currentpid? ∈ dom(pto) ∧
 | TO = pto(currentpid?) ∧
 | pto′ = pto ⊕ { currentpid? ↦ TO′ }
 |_____________________________________________
```

where $\Delta PTO \mathrel{\widehat=} [\ pto,\ pto' : PTO\ ]$. The variable currentpid? gives the identity of the process actually performing the operation.

We can now give final specifications of TestExpiry and Wait (as neither use request identifiers) and updated specifications for the other operations.

$$TestExpiry \mathrel{\widehat=} (TestExpiry_1 \land MPTO) \setminus \Delta TO$$

$$Wait \mathrel{\widehat=} (Wait_1 \land MPTO) \setminus \Delta TO$$

$$SetUpTO_2 \mathrel{\widehat=} (SetUpTO_1 \land MPTO) \setminus \Delta TO$$

$$Delay_2 \mathrel{\widehat=} (Delay_1 \land MPTO) \setminus \Delta TO$$

$$CancelTO_2 \mathrel{\widehat=} (CancelTO_1 \land MPTO) \setminus \Delta TO$$

$$CancelTOReq_2 \mathrel{\widehat=} (CancelTOReq_1 \land MPTO[pid/currentpid?]) \setminus \Delta TO$$

For $CancelTOReq_2$ the identity of the process whose timeout is cancelled is determined by a request identifier rather than being the current process (see below).

Process Activation

When a process is initiated the system sets up its time-out state.

```
┌─ Initiate_TO ──────────────────────────────
│ ΔPTO
│ pid? : PId
├──────────────────
│ pto' = pto ⊕ { pid? ↦ TO_INIT }
└────────────────────────────────────────────
```

When a process terminates the system removes its timeout state.

```
┌─ Terminate_TO ─────────────────────────────
│ ΔPTO
│ pid? : PId
├──────────────────
│ pto' = {pid?} ⩤ pto
└────────────────────────────────────────────
```

Request Identifiers

To complete the time-out system we need to introduce request identifiers and a map that associates a unique request identifier with a process.

$$REQ \mathrel{\widehat{=}} PId \rightarrowtail\!\!\!\!\!\!\mapsto ReqId$$

For SetUpTO and Delay we record the supplied request identifier $reqid?$ in req for the current process.

SetUpReqId

$$\Delta REQ$$
$$pid : PId$$
$$reqid? : ReqId$$

$$reqid? \notin ran(\{ pid \} \lhd req) \land$$
$$req' = req \oplus \{ pid \mapsto reqid? \}$$

where $\Delta REQ \mathrel{\widehat{=}} [\ req,\ req' : REQ\ ]$. The request identifier supplied must not already be in use by any other process. The $reqid?$ is recorded in req for the current process.

On cancels we need to find the process associated with the request identifier.

FindReqId

$$\Delta REQ$$
$$reqid? : ReqId$$
$$pid : PId$$

$$reqid? \in ran(req) \land$$
$$pid = req^{-1}(reqid?)$$

On cancels will also need to delete the entry for the request identifier.

```
DeleteReqId______________________________
| ΔREQ
| pid : PId
|__________________
| req′ = {pid} ⩤ req
|________________________________________
```

With the introduction of request identifiers we have some additional errors. When setting up a reqid?, if it is already in use by some other process, we get an *InvReq* exception.

```
NonUnique!_______________________________
| ≡REQ
| pid : PId
| reqid? : ReqId
| report! : CONDITION
|__________________
| reqid? ∈ ran({pid} ⩤ req) ∧
| report! = InvReq
|________________________________________
```

where ≡REQ ≙ [ΔREQ | req′ = req].

If the reqid? is not found in req we get a *NotFnd* exception.

```
NotFound!________________________________
| ≡REQ
| reqid? : ReqId
|__________________
| reqid? ∉ ran(req) ∧
| report! = NotFnd
|________________________________________
```

We can now complete our specification of operations involving request identifiers.

$$\begin{array}{lcll}
\mathsf{SetUpTO} & \widehat{=} & (\mathsf{SetUpTO}_2 & \wedge\ \mathsf{SetUpReqId}[\mathsf{currentpid?}/\mathsf{pid}]) \\
 & \vee & (\Xi\mathsf{PTO} & \wedge\ \mathsf{NonUnique!}[\mathsf{currentpid?}/\mathsf{pid}]) \\[1em]
\mathsf{Delay} & \widehat{=} & (\mathsf{Delay}_2 & \wedge\ \mathsf{SetUpReqId}[\mathsf{currentpid?}/\mathsf{pid}]) \\
 & \vee & (\Xi\mathsf{PTO} & \wedge\ \mathsf{NonUnique!}[\mathsf{currentpid?}/\mathsf{pid}]) \\[1em]
\mathsf{CancelTO} & \widehat{=} & \ \mathsf{CancelTO}_2 & \wedge\ \mathsf{DeleteReqId}[\mathsf{currentpid?}/\mathsf{pid}] \\[1em]
\mathsf{CancelTOReq} & \widehat{=} & (\mathsf{CancelTOReq}_2 & \wedge\ \mathsf{FindReqId}\ \ \wedge\ \mathsf{DeleteReqId}) \\
 & \vee & (\mathsf{NotFound!} & \wedge\ \Xi\mathsf{PTO})
\end{array}$$

On errors due to request identifiers, the timeout state is not modified. If a process cancels its own timeout ($\mathsf{CancelTO}$) the request identifier for that process is deleted.

When the time for a timeout request expires the system removes all knowledge of the corresponding request identifier.

$$\begin{array}{|l}
\hline
\textsf{Expiry} \\
\quad \Delta\mathsf{REQ} \\
\quad \Xi\mathsf{PTO} \\
\quad \mathsf{pid?} : \mathsf{PId} \\
\hline
\quad \mathsf{timeout}(\mathsf{pto}(\mathsf{pid})) \leq \mathsf{clock}\ \wedge \\
\quad \mathsf{req}' = \{\mathsf{pid?}\} \mathbin{\lhd\!\!\!-} \mathsf{req} \\
\hline
\end{array}$$

where $\Xi\mathsf{PTO} \mathrel{\widehat{=}} [\ \Delta\mathsf{PTO} \mid \mathsf{pto}' = \mathsf{pto}\ ]$.

Start and Retrieve

A Start command may be used to start a transaction at a given time. The transaction runs a given transaction program. It may be associated with a terminal and may have data passed to it. A started transaction may use a Retrieve command to retrieve data passed to it by a Start command. We will represent a transaction by

```
Transaction___________________________
| transid   : TransId
| starttime : Time
| termid    : TermId
| retrdata  : Data
|______________________________________
```

where TransId is the set of names of transaction programs and TermId is the set of terminal identifiers. If a transaction is not associated with a terminal then its termid will be *nil* (i.e. *nil* ∈ Termid). The type Data will not be further refined here but for the moment we can think of it as a sequence of bytes. If there is no data for a transactiou to retrieve then its retrdata will be *nil* (i.e. *nil* ∈ Data).

Transaction and terminal identifiers supplied to a Start command must be in the set of all transaction program names and the set of terminals known to the system, respectively.

```
programnames : ℙ TransId
terminals    : ℙ TermId
```

We associate a unique identifier from the set PId with each transaction in the system. This is so we can distinguish two transactions with the same transid, starttime, termid, and retrdata.

The state of the transaction start/retrieve system is given by

```
TR____________________________________________
|  tr : PId ⇸ Transaction
|  active : ℙ PId
|  retrieved : ℙ PId
|  busy : ℙ TermId
|  clock : Time
|_____________
|
|  active ⊆ dom(tr) ∧
|  retrieved ⊆ active ∧
|  busy = (termid ∘ tr)⦇active⦈ − { nil } ∧
|  (∀pid : active • tr(pid).starttime ≤ clock) ∧
|  active ◁ (termid ∘ tr) ⩥ {nil}  ∈  PId⤔TermId ∧
|  ran(termid ∘ tr) − { nil } ⊆ terminals ∧
|  ran(transid ∘ tr) ⊆ programnames
|_____________________________________________
```

The main component of the state is the map tr which gives the transaction information for each transaction. The active (or running) transactions are a subset of those known (dom(tr)), and the processes whose data has been retrieved must have been active. The bnsy terminals are those currently associated with an active transaction (excluding the special terminal identifier *nil* which signifies there is no terminal attached). The starting time of every active transaction must have already passed. Each actual terminal is associated with at most one active transaction. The terminal and transaction identifiers of transactions mnst be in the sets of tbose known to the system.

The initial state of the transaction system is given by

$$TR_{INIT} \mathrel{\widehat{=}} [\ TR \mid tr = \{\}\ ].$$

A state change is given by

$$\Delta TR \mathrel{\widehat{=}} TR \wedge TR' \wedge \Delta Time$$

The Start command sets up a transaction.

```
Start_0 ──────────────────────────────────────
│ ΔTR
│ Transaction?
│ pid : PId
├──────────────────
│ termid? ∈ terminals ∧
│ transid? ∈ programnames ∧
│ pid ∉ dom(tr) ∧
│ tr′ = tr ∪ { pid ↦ Transaction? } ∧
│ active′ = active ∧
│ retrieved′ = retrieved
└──────────────────────────────────────────────
```

The termid and transid of the new transaction must be members of, respectively, the set of known terminals and the set of known transaction program names. The new transaction is added to the as yet unactivated transactions with a new unique identifier pid.

Aside 1: A Start on the local system causes the current timeout (see Interval Control time-out specification) to be lost. I have chosen not to model this aspect of the operation as it would require adding the time-out state to the above schema. The timeout and start/retrieve operations should be logically separate. □

Aside 2: We will not attempt to model the effect of the nocheck and protect options here. They are to do with the implementation of recovery mechanisms. □

A Start command can cause an exception if the termid? is not one of the available terminals.

```
TermIdErr! ─────────────────────────────
│ ΞTR
│ termid? : TermId
│ report! : Condition
├──────────────────
│ termid? ∉ terminals ∧
│ report! = TermIdErr
└───────────────────────────────────────
```

where ΞTR ≙ [ΔTR | tr′ = tr ∧ active′ = active].

A Start can also cause an exception if the transid is not one of the known transaction program names.

```
TransIdErr! ─────────────────────────────
│ ΞTR
│ transid? : TransId
│ report! : Condition
├───────────────
│ transid? ∉ programnames ∧
│ report! = TransIdErr
└─────────────────────────────────────────
```

Finally, a request identifier is setup on a Start; this may cause an error because it is not unique. The final definition of Start is

$$\begin{array}{lll} \mathsf{Start} & \widehat{=} & (\mathsf{Start}_0 \wedge \mathsf{SetUpReqId} \wedge \mathsf{Success}) \\ & \vee & (\mathsf{TermIdErr!} \wedge \Xi\mathsf{REQ}) \\ & \vee & (\mathsf{TransIdErr!} \wedge \Xi\mathsf{REQ}) \\ & \vee & (\mathsf{NonUnique!} \wedge \Xi\mathsf{TR}) \end{array}$$

Aside: For Start/Retrieve the request identifier is also used as the Temporary Storage queue name under which the data is stored. This implies Start and Retrieve should also modify the Temporary Storage state. I have chosen not to model this as it is an implementation mechanism that should not be visible. □

Activating Transactions

A transaction may be activated by the system if its starttime has passed and if its associated terminal, if it has one, is free. The system action of activating of a transaction is given by

```
Activate_TR ─────────────────────────────────────────────
│ ΔTR
│ pid? : PId
├──────────────
│ pid? ∈ dom(tr) − active ∧
│ tr(pid?).starttime ≤ clock ∧
│ tr(pid?).termid ∉ busy ∧
│ ¬(∃p : active • tr(p).transid = tr(pid?).transid) ∧
│ active′ = active ∪ { pid? } ∧
│ tr′ = tr ∧
│ retrieved′ = retrieved
└─────────────────────────────────────────────────────────
```

The transaction to be activated must be known to the system and not already be active. The transaction's starting time must have passed and its terminal must be free. Only one transaction with a given transid may be active at any one time. The transaction is noted as active.

Deactivation of a process with respect to Interval Control is given by

```
Deactivate_TR ──────────────────────────
│ ΔTR
│ pid? : PId
├──────────────
│ pid? ∈ active ∧
│ tr′ = { pid? } ⩤ tr ∧
│ active′ = active − { pid? } ∧
│ retrieved′ = retrieved − { pid? }
└────────────────────────────────────────
```

The process must have been active. All knowledge of it is removed.

Data Retrieval

An active process may retrieve the data associated with its initiating Start command. After that data has been retrieved, data associated with other transactions, whose starttimes have expired and which have the same termid and transid, may be retrieved. The data from these other transactions is retrieved in starttime order. First we will give the common parts of the Retrieve operation and its associated error actions.

```
ΔTRR______________________________________________________________
| ΔTR
| currentpid? : PId
| Transaction
| retrievable : ℙ PId
|_______________________
| Transaction = tr(currentpid?) ∧
| let possible =
|   dom (tr ▷ {Transaction₁ | termid₁ = termid ∧
|                              transid₁ = transid ∧
|                              starttime₁ ≤ clock} ) in
| retrievable = { p : possible | ∀u : possible •
|                   tr(p).starttime ≤  tr(u).starttime } ∧
| active' = active
|_________________________________________________________________
```

The state of the current transaction is represented by Transaction, that is, transid, termid, starttime and retrdata. A transaction can only possibly retrieve data from a transaction with the same transaction identifier and terminal identifier, whose starting time has expired. Of these it chooses one with a minimal starting time.

The Retrieve operation is given by

```
Retrieve₀─────────────────────────────────────────────
│ ΔTRR
│ data! : Data
├──────────────
│   retrdata ≠ nil
│ ∧ currentpid? ∉ retrieved ⇒
│     data! = retrdata ∧
│     retrieved′ = retrieved ∪ { currentpid? } ∧
│     tr′ = tr
│ ∧ currentpid? ∈ retrieved ⇒
│     (∃p: retrievable • data! = tr(p).retrdata ∧
│                        tr′ = { p } ⩤ tr ∧
│                        retrieved′ = retrieved)
└─────────────────────────────────────────────────────
```

A transaction will first attempt to retrieve its own data; there must have been some supplied when it was started. If the transaction retrieves its own data it is marked as having done so. If a transaction has already retrieved its own data then it may retrieve data from transactions in the set retrievable described above. The transaction whose data was retrieved is deleted.

Aside 1: The current implementation of CICS returns data with equal starttimes in the order in which the corresponding Start commands were issued. No doubt applications may depend on this but the manual does not define the order (nor does it explicitly say it is arbitrary). □

Aside 2: The CICS Start and Retrieve commands have additional parameters: RTransId, RTermId and Queue which are used to pass more data of a specific type. For our specification we will assume that these parameters are passed as part of the retrdata along with a sequence of bytes of normal data. These parameters are really redundant as a structure containing them could be passed as data. □

Aside 3: We have not modelled the "wait" parameter to Retrieve or the time-out on a Retrieve with the "wait" option. □

A Retrieve command can get an exception if there is no data left to be retrieved.

```
EndData! ─────────────────────────────
│ ≡TRR
│ report! : Condition
├──────────────
│ currentpid? ∈ retrieved ∧
│ retrievable = {} ∧
│ report! = EndData
└─────────────────────────────────────
```

where ≡TRR ≙ ΔTRR ∧ ≡TR.

Aside: EndData also occurs on system shutdown. □

A Retrieve can get an exception if no data was supplied on its corresponding Start or if the data of its Start has been retrieved and there is another transaction retrievable by the current transaction for which there was no data supplied.

```
NotFnd! ──────────────────────────────────────────
│ ≡TRR
│ report! : Condition
├──────────────
│ (retrdata = nil
│ ∨ (currentpid? ∈ retrieved ∧
│     ∃p: retrievable • tr(p).retrdata = nil )
│ ) ∧
│ report! = NotFnd
└─────────────────────────────────────────────────
```

The final definition of Retrieve is

$$\text{Retrieve} \mathrel{\widehat{=}} (\text{Retrieve}_0 \wedge \text{Success}) \vee \text{EndData!} \vee \text{NotFnd!}$$

If there are retrievable transactions some of which have data and some of which do not, the above allows the implementation to chose between retrieving data and giving a *NotFnd* exception.

Aside: Exceptions can also occur for the following reasons: input/output errors (IOErr), a dummy temporary storage module is installed in the system (InvTSReq), the format of the data is incorrect (EnvDefErr, LengErr) or there is an invalid parameter (InvReq).

Cancel

A transaction set up by a Start command may be cancelled provided its starttime has not passed.

```
CancelTR₀──────────────────────────────
│ ΔTR
│ pid : PId
├─────────────────
│ pid ∈ dom(tr) ∧
│ tr(pid).starttime > clock
│ tr' = {pid} ⩤ tr
│ active' = active
└──────────────────────────────────────
```

The cancelled transaction is removed from the known transactions. The identity of the transaction to be cancelled (pid) is determined by a request identifier; on cancelling the request identifier is deleted.

$$\begin{aligned}\mathsf{CancelTR} \mathrel{\widehat{=}}\ & (\mathsf{CancelTR}_0 \wedge \mathsf{FindReqId} \wedge \mathsf{DeleteReqId} \wedge \mathsf{Success}) \\ \vee\ & (\mathsf{NotFound!} \wedge \Xi \mathsf{TR})\end{aligned}$$

The cancel operation either cancels a timeout or a start.

$$\mathsf{Cancel} \mathrel{\widehat{=}} \mathsf{CancelTO} \vee \mathsf{CancelTR}$$

The domains of the two operations CancelTO and CancelTR are disjoint. The choice between the two alternatives depends on what operations have taken place previously. For CancelTO, pid must be in the domain of pto. The only operation that achieves this is the timeout $\mathsf{Initiate}_{TO}$; therefore pid must correspond to an active process: pid ∈ active.

For CancelTR, pid must be in the domain of tr and furthermore its starting time must uot have expired. When the Start command corresponding to pid was issued it resulted with pid ∉ active. The only way pid can become active is via a transaction $\mathsf{Activate}_{TR}$, but for a transaction to be activated its starting time must have expired. Therefore, if CancelTR is applicable, the transaction has not been activated. Hence the domains of CancelTO and CancelTR are disjoint.

CICS Message System

Abstract

The following message system is based on the message handling in CICS. The specification itself is an interesting example: it combines states (of input and output devices), and gives a number of examples of the use of the ">>" operator on schemas.

Message Output

We can represent a set of output devices by a mapping from a device name to a sequence of messages that have been ontput to that device.

$$\begin{array}{|l}
\mathsf{NOUT} \\
\hline
\mathsf{noq} : \mathsf{Name} \rightarrowtail\!\!\!\!\!\!+\; \mathsf{seq}\ \mathsf{Message} \\
\hline
\end{array}$$

The operations on outpnt that we will discuss here neither create nor destroy devices.

$$\Delta\mathsf{NOUT} \;\widehat{=}\; [\ \mathsf{NOUT} \wedge \mathsf{NOUT}' \mid \mathsf{dom}\ \mathsf{noq}' = \mathsf{dom}\ \mathsf{noq}\ ]$$

Sending a message to a device simply appends the message to the queue for that device.

$$\begin{array}{|l}
\mathsf{NSend}_0 \\
\hline
\Delta\mathsf{NOUT} \\
\mathsf{n?} : \mathsf{Name} \\
\mathsf{m?} : \mathsf{Message} \\
\hline
\mathsf{noq}' = \mathsf{noq} \oplus \{\ \mathsf{n?} \mapsto \mathsf{noq}(\mathsf{n?}) \frown [\mathsf{m?}]\ \} \\
\hline
\end{array}$$

Multiple Destinations

A message may be sent to a set of destinations.

$$
\begin{array}{|l}
\textsf{NSendM}_0 \\
\hline
\quad \Delta \textsf{NOUT} \\
\quad \textsf{ns?} : \mathbb{P}\ \textsf{Name} \\
\quad \textsf{m?} \ : \textsf{Message} \\
\hline
\quad \textsf{ns?} \subseteq \textsf{dom noq} \land \\
\quad \textsf{noq}' = \textsf{noq} \oplus \{\ \textsf{n} : \textsf{ns?} \bullet \textsf{n} \mapsto \textsf{noq(n)} \frown [\textsf{m?}]\ \} \\
\hline
\end{array}
$$

All the names in ns? must correspond to valid output devices. Each device in n? is sent the message.

<u>Theorem:</u>

Given

$$\textsf{ToSet} \ \widehat{=}\ [\ \textsf{n?} : \textsf{Name};\ \textsf{ns!} : \mathbb{P}\ \textsf{Name} \mid \textsf{ns!} = \{\ \textsf{n?}\ \}\ ]$$

the following equality holds

$$\textsf{NSend}_0 = \textsf{ToSet} >> \textsf{NSendM}_0$$

The schema operator ">>" identifies the outputs (variables ending in "!") of its left operand with the inputs (variables ending in "?") of its right operand; these variables are hidden in the result. All other components are combined together as per schema conjunction ($\land$).

Message Input

We can represent a set of input devices by a mapping from a device name to a sequence of messages yet to be input from that device.

$$\begin{array}{|l}
\text{NIN} \\
\hline
niq : Name \nrightarrow \text{seq } Message \\
\hline
\end{array}$$

The operations on input described here will neither create nor destroy devices.

$$\Delta NIN \mathrel{\widehat{=}} [\ NIN \land NIN' \mid \text{dom } niq' = \text{dom } niq\ ]$$

Receiving a message from a device simply removes it from the head of the input queue for that device.

$$\begin{array}{|l}
\text{NReceive}_0 \\
\hline
\Delta NIN \\
n? : Name \\
m! : Message \\
\hline
m! = head(niq(n?)) \land \\
niq' = niq \oplus \{\ n? \mapsto tail(niq(n?))\ \} \\
\hline
\end{array}$$

Send and Receive

We can define an operation that both sends a message to a device and receives a message from that device.

$$NSendReceive_0 \mathrel{\widehat{=}} NSend_0 \land NReceive_0$$

Combining Input and Output

We will introduce NDEV to describe the combined input and output state for all the devices. If a device can be used for input then it must be able to be used for output.

```
NDEV_____________________________
| NIN
| NOUT
|_______________
| dom niq ⊆ dom noq
|________________________________
```

Input and output operations will preserve the output and input states respectively.

$$\Xi NOUT \mathrel{\widehat{=}} [\ \Delta NDEV \mid NOUT' = NOUT\ ]$$

$$\Xi NIN \mathrel{\widehat{=}} [\ \Delta NDEV \mid NIN' = NIN\ ]$$

where $\Delta NDEV \mathrel{\widehat{=}} NDEV \wedge NDEV'$.

The operations on the combined state are

$$NSend \mathrel{\widehat{=}} NSend_0 \wedge \Xi NIN$$

$$NSendM \mathrel{\widehat{=}} NSendM_0 \wedge \Xi NIN$$

$$NReceive \mathrel{\widehat{=}} NReceive_0 \wedge \Xi NOUT$$

$$NSendReceive \mathrel{\widehat{=}} NSendReceive_0 \wedge \Delta NDEV$$

Logical Names

Rather than work with actual (physical) device names, as we have up until this point, we would like to work with logical names that are mapped into physical device names. We use the following mapping from logical names to physical names.

```
LtoP___________________________________
|  ltop : LName ⇸ Name
|______________________________________
```

None of the operations discussed here modify the mapping from logical names to physical names hence we will use

```
ΞLtoP  ≙  [ LtoP ∧ LtoP'  |  LtoP' = LtoP ]
```

If a logical name actually corresponds to a device we perform the operation on that device, otherwise we use the device with physical name `console`.

```
MapName______________________________________________
|  ΞLtoP
|  dev : Name ⇸ seq Message
|  ln? : LName
|  n!  : Name
|_________________
|
|  ln? ∈ dom(ltop⨾dev)   ⇒   n! = ltop(ln?)   ∧
|  ln? ∉ dom(ltop⨾dev)   ⇒   n! = console
|____________________________________________________
```

The operations on a single device become

```
LSend          ≙  MapName[noq/dev] >> NSend

LReceive       ≙  MapName[niq/dev] >> NReceive

LSendReceive ≙  MapName[niq/dev] >> NSendReceive
```

Multiple Logical Destinations

To send a message to a set of logical names we need to map the set of logical names into physical names. If none of the logical names correspond to a device we send the message to the device with physical name `console`.

```
MapSet
 ΞLtoP
 lns? : ℙ LName
 ns!  : ℙ Name
 NOUT
 ──────────────
 let  names = ltop⦇lns?⦈ ∩ dom noq  in
   names = {} ⇒ ns! = { console }  ∧
   names ≠ {} ⇒ ns! = names
```

The operation to send a message to a set of logical devices is

```
LSendM  ≙  MapSet >> NSendM
```

<u>Theorem:</u>

Given

```
ToSetL  ≙  [ ln? : LName; lns! : ℙ LName  |  lns! = { ln? } ]
```

the following equality holds

```
LSend  =  ToSetL >> LSendM
```

Domains of the Operations

In practice we would like all the operations to be total (defined for all inputs). Unfortunately the operations as defined are not total. If a name (or a set of names) does not correspond to an actual device then the name will be translated to the special device console; if the console does not exist the operation is not defined. For the output operations ensuring that the console exists is a sufficient pre-condition for the operation to be defined. (We will also need this pre-condition for input.)

$$Pre \mathrel{\widehat{=}} [\ NDEV;\ LtoP;\ m? : Message \mid console \in \mathrm{dom}\ niq\ ]$$

Remember that $\mathrm{dom}\ niq \subseteq \mathrm{dom}\ noq$ so $console \in \mathrm{dom}\ noq$.

Theorems:

$$Pre \implies pre\ LSend$$

$$Pre \implies pre\ LSendM$$

For the input operations we need the additional requirement that the queue of messages yet to be input on the device is not empty.

$$PreIn \mathrel{\widehat{=}} [\ Pre;\ n? : Name \mid niq(n?) \neq [\,]\ ]$$

Theorems:

$$MapName[niq/dev] \gg PreIn \implies pre\ LReceive$$

$$MapName[niq/dev] \gg PreIn \implies pre\ LSendReceive$$

Acknowledgement

This specification is based on a message system specified by David Renshaw of IBM (U. K.) Laboratories, Hursley, England.

Z Reference Card
Mathematical Notation
Version 2.1

Programming Research Group
Oxford University

1. Definitions and declarations.

Let x, x_k be identifiers and T, T_k sets.

$LHS \mathrel{\hat=} RHS$	Definition of LHS as syntactically equivalent to RHS.
$x:\ T$	Declaration of x as type T.
$x_1:\ T_1;\ x_2:\ T_2;\ \ldots\ ;\ x_n:\ T_n$	List of declarations.
$x_1,\ x_2,\ \ldots\ ,\ x_n\ :\ T$	$\mathrel{\hat=} x_1:T;\ x_2:T;\ \ldots\ ;\ x_n:T.$

2. Logic.

Let P, Q be predicates and D declarations.

$\neg\ P$	Negation: "not P".
$P \wedge Q$	Conjuuctiou: "P and Q".
$P \vee Q$	Disjunction: "P or Q".
$P \Rightarrow Q$	Implication: "P implies Q" or "if P then Q".
$P \Leftrightarrow Q$	Equivalence: "P is logically equivalent to Q".
$\forall\ x\ :\ T \bullet P$	Universal quantification: "for all x of type T, P holds".
$\exists\ x\ :\ T \bullet P$	Existential quantification: "there exists an x of type T such that P".
$\exists!\ x\ :\ T \bullet P_x$	Unique existence: "there exists a unique x of type T such that P". $\mathrel{\hat=} (\exists\ x\ :\ T \bullet P_x \wedge \neg(\exists y:T \mid y \neq x \bullet P_y))$
$\forall\ x_1:T_1;\ x_2:T_2;\ \ldots\ ;\ x_n:T_n \bullet P$	"For all x_1 of type T_1, x_2 of type $T_2, \ldots,$ and x_n of type T_n, P holds.
$\exists\ x_1:T_1;\ x_2:T_2;\ \ldots\ ;\ x_n:T_n \bullet P$	Similar to $\forall$.
$\exists!\ x_1:T_1;\ x_2:T_2;\ \ldots\ ;\ x_r:T_n \bullet P$	Similar to $\forall$.
$\forall\ D \mid P \bullet Q$	$\mathrel{\hat=} (\forall\ D \bullet P \Rightarrow Q)$.
$\exists\ D \mid P \bullet Q$	$\mathrel{\hat=} (\exists\ D \bullet P \wedge Q)$.
$t_1 = t_2$	Equality between terms.
$t_1 \neq t_2$	$\mathrel{\hat=} \neg(t_1 = t_2)$.

3. Sets.

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

$t \in S$	Set membership: "t is an element of S".
$t \notin S$	$\mathrel{\hat=} \neg(t \in S)$.
$S \subseteq T$	Set inclusion: $\mathrel{\hat=} (\forall\ x\ :\ S \bullet x \in T)$.
$S \subset T$	Strict set inclusion: $\mathrel{\hat=} S \subseteq T \wedge S \neq T$.
$\{\}$	The empty set.
$\{\ t_1,\ t_2,\ \ldots\ ,\ t_n\ \}$	The set containing $t_1, t_2, \ldots$ and t_n.
$\{\ x\ :\ T \mid P\ \}$	The set containing exactly those x of type T for which P holds.
$(t_1,\ t_2,\ \ldots\ ,\ t_n)$	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 kth component is of type T_k.
$\{\ x_1:T_1;\ x_2:T_2;\ \ldots\ ;\ x_n:T_n \mid P\ \}$	The set of n-tuples $(x_1,\ x_2,\ \ldots\ ,\ x_n)$ with each x_k of type T_k such that P holds.

$\{ D \mid P \bullet t \}$	The set of t's such that given the declarations D, P holds.
$\{ D \bullet t \}$	$\widehat{=} \{ D \mid true \bullet t \}$.
$\mathbb{P}\ S$	Powerset: the set of all subsets of S.
$\mathbb{F}\ S$	Set of finite subsets of S: $\widehat{=} \{ T : \mathbb{P}\ S \mid T \text{ is finite} \}$.
$S \cap T$	Set intersection: given $S, T : \mathbb{P}\ X$, $\widehat{=} \{ x:X \mid x \in S \land x \in T \}$.
$S \cup T$	Set union: given $S, T : \mathbb{P}\ X$, $\widehat{=} \{ x:X \mid x \in S \lor x \in T \}$.
$S - T$	Set difference: given $S, T : \mathbb{P}\ X$, $\widehat{=} \{ x:X \mid x \in S \land x \notin T \}$.
$\bigcap SS$	Distributed set intersection: given $SS : \mathbb{P}\ (\mathbb{P}\ X)$, $\widehat{=} \{x:X \mid (\forall S:SS \bullet x \in S)\}$.
$\bigcup SS$	Distributed set union: given $SS : \mathbb{P}\ (\mathbb{P}\ X)$, $\widehat{=} \{x:X \mid (\exists S:SS \bullet x \in S)\}$.
$\lvert S \rvert$	Size (number of distinct elements) of a finite set.

4. Numbers.

$\mathbb{N}$	The set of natural numbers (non-negative integers).
$\mathbb{N}^+$	The set of strictly positive natural numbers: $\widehat{=} \mathbb{N} - \{ 0 \}$.
$\mathbb{Z}$	The set of integers (positive, zero and negative).
$m \ldots n$	The set of integers between m and n inclusive: $\widehat{=} \{ k:\mathbb{Z} \mid m \leq k \land k \leq n \}$.
$\min S$	Minimum of a set, $S : \mathbb{F}\ \mathbb{N}$. $\min S \in S \land (\forall x : S \bullet x \geq \min S)$.
$\max S$	Maximum of a set, $S : \mathbb{F}\ \mathbb{N}$. $\max S \in S \land (\forall x : S \bullet x \leq \max S)$.

5. 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: $\widehat{=} (x, y) \in R$.
$x \mapsto y$	$\widehat{=} (x, y)$
$\{ x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n \}$	The relation $\{ (x_1, y_1), \ldots, (x_n, y_n) \}$ relating x_1 to y_1, ..., and x_n to y_n.
$\mathrm{dom}\ R$	The domain of a relation: $\widehat{=} \{x:X \mid (\exists y:Y \bullet x\ R\ y)\}$.
$\mathrm{rng}\ R$	The range of a relation: $\widehat{=} \{y:Y \mid (\exists x:X \bullet x\ R\ y)\}$.
$R_1 \mathbin{;} R_2$	Forward relational composition: given $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$, $\widehat{=} \{ x:X;\ z:Z \mid (\exists y:Y \bullet x\ R_1\ y \land y\ R_2\ z)\}$.
$R_1 \circ R_2$	Relational composition: $\widehat{=} R_2 \mathbin{;} R_1$.
R^{-1}	Inverse of relation R: $\widehat{=} \{ y:Y;\ x:X \mid x\ R\ y \}$.
$\mathrm{id}\ X$	Identity function on the set X: $\widehat{=} \{ x : X \bullet x \mapsto x \}$.
R^k	The relation R composed with itself k times: given $R : X \leftrightarrow X$, $R^0 \widehat{=} \mathrm{id}\ X$, $R^{k+1} \widehat{=} R^k \circ R$.
R^*	Reflexive transitive closure: $\widehat{=} \bigcup \{ n : \mathbb{N} \bullet R^n \}$.
R^+	Non-reflexive transitive closure: $\widehat{=} \bigcup \{ n : \mathbb{N}^+ \bullet R^n \}$.
$R(\lvert S \rvert)$	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 $S: \mathbb{P}\ X$, $\hat{=} \{x:X; y:Y \mid x \in S \land x\ R\ y\}$.
$S ⩤ R$	Domain subtraction: given $S: \mathbb{P}\ X$, $\hat{=} (X - S) \lhd R$.
$R \rhd T$	Range restriction to T: given $T: \mathbb{P}\ Y$, $\hat{=} \{x:X; y:Y \mid x\ R\ y \land y \in T\}$.
$R ⩥ T$	Range subtraction of T: given $T: \mathbb{P}\ Y$, $\hat{=} R \rhd (Y - T)$.
$R_1 \oplus R_2$	Overriding: given $R_1, R_2 : X \leftrightarrow Y$, $\hat{=} (\mathrm{dom}\ R_2 ⩤ R_1) \cup R_2$.

6. 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 defined above for relations also apply to functions.

$X ⇸ Y$	The set of partial functions from X to Y: $\hat{=} \{ f: X \leftrightarrow Y \mid$ $(\forall x: \mathrm{dom}\ f \bullet$ $(\exists! y: Y \bullet x\ f\ y)) \}$.
$X \rightarrow Y$	The set of total functions from X to Y: $\hat{=} \{ f: X ⇸ Y \mid \mathrm{dom}\ f = X \}$.
$X ⤔ Y$	The set of one-to-one partial functions from X to Y: $\hat{=} \{ f: X ⇸ Y \mid$ $(\forall y: \mathrm{rng}\ f \bullet$ $(\exists! x: X \bullet x\ f\ y)) \}$.
$X \rightarrowtail Y$	The set of one-to-one total functions from X to Y: $\hat{=} \{ f: X ⤔ Y \mid \mathrm{dom}\ f = X \}$.
$f\ t$	The function f applied to t.

$(\lambda\ x : X \mid P \bullet t)$

Lambda-abstraction:
the function that given an argument x of type X such that P holds the result is t.
$\hat{=} \{ x: X \mid P \bullet x \mapsto t \}$.

$(\lambda\ x_1: T_1; \ \ldots\ ; x_n: T_n \mid P \bullet t)$

$\hat{=} \{x_1:T_1; \ \ldots\ ; x_n:T_n \mid P \bullet (x_1, \ \ldots\ , x_n) \mapsto t \}$.

7. Orders.

partial_order X

The set of partial orders on X.
$\hat{=} \{ R: X \leftrightarrow X \mid \forall x, y, z: X \bullet$
$x\ R\ x \land$
$x\ R\ y \land y\ R\ x \Rightarrow x=y \land$
$x\ R\ y \land y\ R\ z \Rightarrow x\ R\ z$
$\}$.

totel_order X

The set of total orders on X.
$\hat{=} \{ R: \text{pertial\_order}\ X \mid$
$\forall x, y: X \bullet$
$x\ R\ y \lor y\ R\ x$
$\}$.

monotonic $X <_X$

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

8. Sequences.

seq X	The set of sequences whose elements are drawn from X: $\widehat{=} \ \{ A : \mathbb{N}^{+} \nrightarrow X \mid \mathrm{dom}\ A = 1..\lvert A\rvert \}$.
$\lvert A\rvert$	The length of sequence A.
[ ]	The empty sequence { }.
$[a_1, \ldots, a_n]$	$\widehat{=} \ \{ 1 \mapsto a_1, \ldots, n \mapsto a_n \}$.
$[a_1, \ldots, a_n] \frown [b_1, \ldots, b_m]$	Concatenation: $\widehat{=} \ [a_1, \ldots, a_n, b_1, \ldots, b_m]$, $[\,] \frown A = A \frown [\,] = A$.
head A	$\widehat{=} \ A(1)$.
last A	$\widehat{=} \ A(\lvert A\rvert)$.
tail $[x] \frown A$	$\widehat{=} \ A$.
front $A \frown [x]$	$\widehat{=} \ A$.
rev $[a_1, a_2, \ldots, a_n]$	Reverse: $\widehat{=} \ [a_n, \ldots, a_2, a_1]$, rev $[\,] = [\,]$.
$\frown/AA$	Distributed concatenation: given AA : seq(seq(X)), $\widehat{=} \ AA(1) \frown \ldots \frown AA(\lvert AA\rvert)$, $\frown/[\,] = [\,]$.
$⨟/AR$	Distributed relational composition: given AR : seq $(X \leftrightarrow X)$, $\widehat{=} \ AR(1) ⨟ \ldots ⨟ AR(\lvert AR\rvert)$, $⨟/[\,] = \mathrm{id}\ X$.
<u>disjoint</u> AS	Pairwise disjoint: given AS: seq $(\mathbb{P}\ X)$, $\widehat{=} \ (\forall\ i, j : \mathrm{dom}\ AS \bullet i \neq j \Rightarrow AS(i) \cap AS(j) = \{\})$.
AS <u>partitions</u> S	$\widehat{=}$ disjoint AS $\wedge \bigcup \mathrm{ran}\ AS = S$.
A <u>in</u> B	Contiguous subsequence: $\widehat{=} \ (\exists C, D : \mathrm{seq}\ X \bullet C \frown A \frown D = B)$.
squash f	Convert a function, $f : \mathbb{N} \nrightarrow X$, into a sequence by squashing its domain. squash {} = [ ], and if $f \neq \{\}$ then squash f = $[f(i)] \frown \mathrm{squash}(\{i\} ⩤ f)$ where $i = \min(\mathrm{dom}\ f)$ e.g. squash $\{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\} = [A, B, C]$
$S \upharpoonleft A$	Restrict the sequence A to those items whose index is in the set S: $\widehat{=} \ \mathrm{squash}(S \lhd A)$
$A \upharpoonright T$	Restrict the range of the sequence A to the set T: $\widehat{=} \ \mathrm{squash}(A \rhd T)$.

9. Bags.

bag X	The set of bags whose elements are drawn from X: $\widehat{=} \ X \nrightarrow \mathbb{N}^{+}$ A bag is represeuted by a function that maps each element in the bag onto its frequency of occurrence in the bag.
$[\![\,]\!]$	The empty bag { }.
$[\![x_1, x_2, \ldots, x_n]\!]$	The bag containing x_1, x_2, ... and x_n with the frequency they occur in the list.
items s	The bag of items contained in the sequence s: $\widehat{=} \ \{ x : \mathrm{rng}\ s \bullet x \mapsto \lvert \{ i : \mathrm{dom}\ s \mid s(i) = x \} \rvert \}$

Z Reference Card
Schema Notation
[For details see "Schemas in Z"]

Programming Research Group
Oxford University

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

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

or horizontally, for the same example

$S \mathrel{\widehat=} [\ x\colon \mathbb{N};\ y\colon \operatorname{seq} \mathbb{N} \mid x \leq |y|\ ].$

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

$(\forall S \bullet y \neq [\,]) \mathrel{\widehat=} (\forall x\colon\mathbb{N};\ y\colon \operatorname{seq} \mathbb{N} \mid x \leq |y| \bullet y \neq [\,]).$

tuple S The tuple formed of a schema's variables.

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

Inclnsion 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 the same type) and the predicates of S and T are conjoined. e.g.

```
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. $(S \mid x>0)$ is $[x\colon\mathbb{N};y\colon\operatorname{seq} \mathbb{N} \mid x\leq|y| \wedge x>0]$.

S ; D The schema S with the declarations D merged with the declarations of S. e.g. $(S\ ;\ z : \mathbb{N})$ is $[\ x, z\colon\mathbb{N};\ y\colon\operatorname{seq} \mathbb{N} \mid x\leq|y|\ ]$

S[new/old] Renaming of components: the schema S with the component old renamed to new in its declaration and every free use of that old within the predicate. e.g. S[z/x] is $[\ z\colon\mathbb{N};\ y\colon\operatorname{seq} \mathbb{N} \mid z \leq |y|\ ]$ and S[y/x, x/y] is $[\ y\colon\mathbb{N};\ x\colon\operatorname{seq} \mathbb{N} \mid y \leq |x|\ ]$

Decoration Decoration with subscript, superscript, prime, etc.; systematic renaming of the variables declared in the schema. e.g. S' is $[x'\colon\mathbb{N};\ y'\colon\operatorname{seq} \mathbb{N} \mid x'\leq|y'|]$

¬S The schema S with its predicate part negated. e.g. ¬S is $[x\colon\mathbb{N};\ y\colon\operatorname{seq} \mathbb{N} \mid \neg(x\leq|y|)]$

S ∧ T The schema formed from schemas S and T by merging their declarations (see inclusion above) and and'ing their predicates. Given $T \mathrel{\widehat=} [x\colon \mathbb{N};\ z\colon \mathbb{P}\,\mathbb{N} \mid x \in z]$, $S \wedge T$ is

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

$S \vee T$ The schema formed from schemas S and T by merging their declarations and or'ing their predicates. e.g. $S \vee T$ 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 $\mathsf{pred}\ S \Rightarrow \mathsf{pred}\ T$ as the predicate. e.g. $S \Rightarrow T$ is similar to $S \wedge T$ and $S \vee T$ except the predicate contains an "$\Rightarrow$" rather than an "$\wedge$" or an "$\vee$".

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking $\mathsf{pred}\ S \Leftrightarrow \mathsf{pred}\ T$ as the predicate. e.g. $S \Leftrightarrow T$ the same as $S \wedge T$ with "$\Leftrightarrow$" in place of the "$\wedge$".

$S \setminus (v_1, v_2, \ldots, v_n)$
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 $[y{:}\mathrm{seq}\ \mathbb{N} \mid (\exists x{:}\mathbb{N} \cdot x \leq |y|)]$

A schema may be specified instead of a list of variables; in this case the variables declared in that schema are hidden.
e.g. $(S \wedge T) \setminus S$ 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)
└──────────────────────
```

The list of variables may be replaced by a schema as for hiding; the variables declared in the schema are used for the projection.

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

undashed	state before the operation,
dashed	state after the operation,
ending in "?"	inputs to the operation, and
ending in "!"	outputs from 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

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

pre S is

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

post S — Postcondition: this is similar to precondition except all the state before components (undashed) and inputs (ending in "?") are hidden.

S ⊕ T — Overriding:
$\widehat{=}\ (S \land \neg \mathrm{pre}\ T) \lor T.$
e.g. given S above and

T

$$\begin{array}{|l}
x?,\ s,\ s' : \mathbb{N} \\
\hline
s < x? \land s' = s
\end{array}$$

S ⊕ T is

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

The predicate can be simplified:

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

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

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

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

S >> 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 "?") that have the same basename.