Applying formal specification to the development of software in industry

Ian Hayes

February 1985

Abstract This chapter 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 are outlined, with particular reference to their application to an existing piece of software. The specification process itself is described and a sample specification presented.

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

Problems with the specification techniques themselves, which arose in applying the techniques to a commercial transaction processing system, are outlined.

1 Introduction

Oxford University and IBM (UK) 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, used so far within a research environment, to large-scale software in an industrial environment. This chapter 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 ap-

plications involving large numbers of terminals (thousands) and very large data bases (requiring gigabytes). The CICS General Information manual [IBM80b] 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. 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, transaction 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 assembly language 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 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, Assembly language 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 to 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.

2 Uses of formal specification

The work reported in this chapter deals with the specification of parts of an 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 [Sta82].) 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 validation 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

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

In the longer term the specification can be used

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

Reimplementation may involve a new machine architecture, programming language or operating system, or a restructuring to take advantage of multiprocessor 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. These developments 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 among the people involved.

Experimentation with specification provides a 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 informal specifications and in the detail of implementation.

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

3 The specification process

The starting point for our specification work was the CICS command-level application programmer's reference manual [IBM80a]. 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. Questions can arise for the following reasons:

- the manual is incomplete or vague;
- the manual is not explicit as to whether *possible* special cases are treated normally or not;
- the manual is itself inconsistent; or
- 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 were either 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 and style, and who 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 had 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 a revision of the specification, which led to further questions and further review, and so on.

3.1 Notation

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 specification readable, 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 because there are too many different facets to understand at 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, because 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.

4 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 [IBM76]. 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 8. The specification given here is the final product of the specification process described in the previous section.

The *syntax* of the CICS commands depends, of course, on the environment in which they are written. Our notation below is intended to be uncommitted, but explicit enough to indicate exactly which command is meant.

4.1 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

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

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

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

$$\mathit{Ignore\ Condition}(c = z)$$

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 this is not a condition that can arise from the execution of a command; rather, it provides a mechanism for providing a catchall error handler for conditions that are not explicitly handled.

We do not list all the possible exceptional conditions here.

$$\mathit{CONDITION} ::= \mathit{success} \mid \mathit{error} \mid \ldots$$

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* is not discussed in detail here. For each programming language supported by CICS it has a slightly different meaning, but for all of the languages an action is represented by a label which is given control. There are four 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.

$$\mathit{ACTION} ::= \mathit{nil} \mid \mathit{abort} \mid \mathit{wait} \mid \mathit{system} \mid \ldots$$

4.2 The state

The state of the exception controlling system can be defined by the following schema:

Exceptions

$$Handler : CONDITION \pfun ACTION$$

$$Handler(success) = nil$$

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

$$Default : CONDITION \to ACTION$$

$$Default(error) = abort \land$$
$$\mathrm{ran}(Default) = \{nil, abort, wait\}$$

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 the following schema:

Initial

$$Exceptions$$

$$Handler = \{success \mapsto nil\}$$

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

$$Handle\ Condition(c = x, a = y)$$
$$Ignore\ Condition(c = z)$$

are executed, then the final state will satisfy

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

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

4.3 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 '$\Delta Exceptions$':

$$\Delta Exceptions$$
$$Exceptions$$
$$Exceptions'$$

Exceptions represents the state of the exception handling system before an operation and $Exceptions'$ the state after.

The operation *Handle Condition* is used to set up the action, $a?$, to be performed on a particular exceptional condition, $c?$. It is defined by the following schema:

$$HandleCondition$$
$$\Delta Exceptions$$
$$c? : CONDITION$$
$$a? : ACTION$$
$$c? \neq success \land a? \notin \{nil, abort, wait\} \land$$
$$Handler' = Handler \oplus$$
$$\{c? \mapsto (\textbf{if } a? = system \textbf{ then } Default(c?) \textbf{ else } a?)\}$$

The first predicate gives the precondition 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 command

$$Handle\ Condition(c = x, a = system)$$

is executed in the initial state and $Default(x) = wait$, the resulting state will satisfy

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

The actual *Handle Condition* command accepts a set of condition–action pairs, rather than just a single pair as shown above. However, the effect of the command for each pair is as described above, so we will not bother to show the full command. Similarly, the *Ignore Condition* command accepts a set of conditions, but we only bother to show its effect for a single condition here.

The operation to specify that an exceptional condition is to be ignored is given by the following schema:

IgnoreCondition

$$\Delta Exceptions$$
$$c? : CONDITION$$

$$c? \neq success$$
$$Handler' = Handler \oplus \{c? \mapsto nil\}$$

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

4.4 Exception checking

Exception handling can take place on any CICS command except *HandleCondition* and *IgnoreCondition* 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 or not all exceptions are to be handled for the execution of just that command. In describing the checking process we include the Boolean variable *handle*? to indicate this. The following defines the (complex) exception checking mechanism that is included in the definition of each operation (other than *Handle Condition* and *Ignore Condition*):

ExceptionCheck

$$Exceptions$$
$$handle? : Boolean$$
$$c? : CONDITION$$
$$a! : ACTION$$

$$a! = \textbf{if } handle? = False \textbf{ then } nil$$
$$\quad \textbf{else if } c? \in \mathrm{dom}\, Handler \textbf{ then } Handler(c?)$$
$$\quad \textbf{else if } Default(c?) \neq abort \textbf{ then } Default(c?)$$
$$\quad \textbf{else if } error \in \mathrm{dom}\, Handler \textbf{ then } Handler(error)$$
$$\quad \textbf{else } abort$$

If exceptions are not being handled for the command ($handle? = False$) 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 is *abort*.

5 Questions raised

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 those 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 precision required, questions are raised during the specification process that are not answered by referring 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 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 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.

5.1 Exceptional conditions

We 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 of 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

$$\begin{array}{l} \mathit{Handle\ Condition}(c = x, a = y) \\ \quad \vdots \\ \mathit{Handle\ Condition}(c = x, a = \mathit{system}) \end{array}$$

return the handler to the initial state?

The reader is invited to try to answer these questions from the manual entry given in Appendix 8 and then from the specification given in Section 4.1. We 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 new value for the $\mathit{Handler}'$ in the case when $a? = \mathit{system}$ was

$$\mathit{Handler}' = \{c?\} \lhd \mathit{Handler}$$

That is, if the action specified as an input is *system* then the entry for the condition $c?$ is removed from the handler ($c? \notin \mathrm{dom}\, \mathit{Handler}'$). In the final model the new value of the $\mathit{Handler}'$ in this case is

$$\mathit{Handler}' = \mathit{Handler} \oplus \{c? \mapsto \mathit{Default}(c?)\}$$

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

To see the effect of the difference we need to look at the *Exception Check* mechanism given in Section 4.1. If we use the second line above, then the action when the exception $c?$ occurs is $\mathit{Default}(c?)$ (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 is $\mathit{Default}(c?)$ unless $\mathit{Default}(c?)$ is *abort* and $\mathit{error} \in \mathrm{dom}\, \mathit{Handler}$, in which case the action is $\mathit{Handler}(\mathit{error})$. 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 was 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, i.e.

$$\begin{array}{l} \mathit{Handle\ Condition}(c = x, a = y) \\ \quad \vdots \\ \mathit{Handle\ Condition}(c = x, a = \mathit{system}) \end{array}$$

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 is used on an occurrence of condition x, but after, the action *Default*(x) (i.e. *abort*) is 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 handle condition x temporarily for the duration of the statements between the *Handle Condition* commands. This form of operation is more what those 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 so that it becomes 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 with the manual, from a person experienced in formal specification. It is important that the reviewer should read 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.

5.2 Interval control

As another example we 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 concerned with starting new transactions at specified times, and those concerned 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, because time-outs would not be affected by a transaction start.

This anomaly is interesting because 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 because 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.

5.3 Interaction between modules

As an example of an interaction between two CICS modules we consider an interaction between exceptional conditions and temporary storage. When temporary storage is exhausted it can raise the exceptional condition *nospace*. This is 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 was ignored.

Interactions between modules of the system are pinpointed during the formal specification process (just as they would be in an implementation) because an operation from one module needs access to the state components of another. Any such interactions discovered during the specification process

should be examined closely as they may indicate a breakdown in the modular structure of the system.

6 Problems with specification

In this section we examine the problems encountered in applying the formal specification techniques. This is in contrast to the previous section, in which we concentrated on the system being specified. The problems encountered in applying specification techniques can be split into the following categories:

- 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.

6.1 Communication problems

As a specification group from a university working with a commercial development laboratory we faced a communications problem. Each party has its 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 that each party learn 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 it is necessary 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 educational benefit has been more to the advantage 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).

6.2 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. It is most important that 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 specified object do not obscure 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 referred to 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.

6.3 Technical problems

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

- putting the module specifications together to provide a specification of the system as a whole;
- specifying parallelism;
- specifying recovery on system failures; and
- specifying distributed systems.

We shall briefly discuss each of 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 module specifications together. Each module has state components and a set of operations that work on those state components. Putting the modules together amounts to 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 as follows:

- avoiding name clashes when the modules were combined;
- specifying the effect on the whole system state of an operation defined within a module of the system; and

- 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.

7 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 is 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 main short-term benefits that are obtained by applying formal specification techniques to existing software are the questions that are raised during the specification process. 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 which incorporates 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 [Mor83].)

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 the 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 when 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.

Acknowledgements I would like to thank IBM for their permission to publish this chapter and reproduce part of one of their manuals as an appendix. 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 and exceptional conditions. Paul Fertig, Roger Gimson, John Nicholls and Bernard Sufrin gave useful comments on this chapter. 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.

8 Appendix: exceptional conditions manual

The following is an extract of the manual entry for exceptional conditions taken from [IBM80a].

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 it. Usually, this default action is to terminate the task abnormally.

However, to prevent abnormal 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 commands, 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 *NOHAN-*

DLE 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 if 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.

References

[IBM76] IBM Corporation. *OS PL/I Checkout and Optimising Compilers: Language reference manual*, 1976.

[IBM80a] IBM Corporation. *CICS/OS/VS Version 1 Release 5, Application Programmer's Reference Manual (Command level)*, 1980.

[IBM80b] IBM Corporation. *CICS/VS General Information*, 1980.

[Mor83] C.C. Morgan. Using mathematics in user manuals. Distributed Computing Project technical report, Programming Research Group, Oxford University, 1983.

[Sta82] J. Staunstrup, editor. *Program Specification: Proceedings of a Workshop, Aarhus, Denmark (August 1981)*, volume 134 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.