A Formal Specification of the QMC Message System: The Underlying Abstract Model*

W. T. ROBERTS
Department of Computer Science, Queen Mary College, London E1 4NS

This report presents an algebraic specification of the Queen Mary College (QMC) Message System, the first part of a complete specification of the user interface. There are few examples in the literature of how to go about the task of constructing an algebraic, axiomatic specification, so this paper also presents the method employed by the author to arrive at the stated specification. A brief example is given of the use of the specification in the description of a user interface, and various technical and implementation issues are discussed.

Received September 1986, revised January 1987

1. INTRODUCTION

Formal specification can have many different purposes. It may be intended as a way of documenting design decisions, of communicating between designers, of communicating from designer to programmer, even of communicating from designer to machine in the case of executable specifications.

In this paper, formal specification is seen as a declaration of intent: writing down as precisely as possible what the designer has in mind. The result is then useful for communication between designers and for recording design decisions. A technique is presented which assists a designer to record his or her ideas in the form of an abstract algebra. It so happens that ways exist of executing such axiomatic specifications, but this is a bonus and not the primary goal of the process.

Section 2 gives some mathematical background to this work, mostly definitions and technicalities that may be omitted on first reading. Section 3 briefly summarises the methodology and the example problem, then Section 4 gives the first worked example, the simplest part of the QMC Message System. The more difficult part of the QMC Message System is specified in Section 5, which introduces some more techniques within the methodology and a much larger example. Section 6 revises the work of Section 4 and introduces some more functionality, requiring yet more new techniques, and then Section 7 combines the two specifications. Section 8 discusses three applications of this work, including an example of the use of the specification in the formal description of a user interface, Section 9 considers a number of issues relating to the technique described, and finally Section 10 concludes.

2. MATHEMATICAL INTRODUCTION

This section is only necessary for those who want some flavour of the mathematics on which this specification method is based.

An abstract algebraic theory consists of a signature and a set of axioms. A signature is a set of function names, each name having an associated arity. The signature corresponds approximately to a set of function declarations in a typed language such as Pascal; in Pascal, a function declaration gives the name of the function together with the number and types of its arguments and the type of its result; in abstract algebra the technical word for 'type' is sort and the arity of a function name is a list of the sorts of its arguments and the sort of the result. Henceforth we will use the word type instead of sort in order to conform to common computer science usage. When specifying an abstract data type we are defining the values of a new type and the functions which can be applied to those values. Typically we will be defining a single new type with the help of several existing ones; this new type will be called the type of interest or TOI for short.

A term is an expression made from the function names with an object of the appropriate type in place of each argument. In particular, any term of the correct type can be used as an argument. This is similar to the composition of functions in Pascal. If the term does not contain variables (see below) it is called a ground term and is effectively a name for a value of the result type associated with the outermost function name. The axioms of a theory are statements that the value of a particular term is equal to the value of another. We allow the use of typed variables in axioms, meaning that the statement is true when any value or term of the appropriate type is substituted for the variable throughout the formula. Other ways of saying this are that such an axiom is really an axiom schema, or that the variables are implicitly universally quantified over the set of values of the appropriate type. We do not normally bother to state the type of each variable since it can be inferred from the context in which it appears.

A signature becomes an abstract algebra when carrier sets are associated with each type and correctly typed functions are associated with each function name in the signature. The axioms of a theory do not necessarily hold in all possible algebras with the correct signature: the functions specified might not behave in the way the axioms require. If all the axioms are true in the algebra, then the algebra satisfies the theory. We are not interested in algebras which fail to satisfy the theories we construct; fortunately, there is a very general way of obtaining suitable carrier sets and functions. First we take the word algebra in which the carrier set for the type of interest is the set of all possible ground terms having that type and the result of each function which has result type TOI is defined to be the term obtained by writing the function
name followed by its arguments. This is not usually correct as it stands, so we form the equivalence relation on terms implied by the axioms; two terms are the same iff the axioms state that they are the same. We then take a quotient of the word algebra by this equivalence relation and arrive at the desired algebra.

An abstract algebraic specification is an (abstract algebraic) theory; a signature and a set of axioms. An implementation of such a specification is an abstract algebra which satisfies the theory. An implementation should also provide an equality predicate for the type of interest: it is permissible for the implementation to have several different representations for the same object (e.g. +0 and −0 in the ones-complement binary representation of the integers), provided that a function is provided for ‘ = ’ which returns True when comparing different representations of the same object. This equality function is used when checking axioms which relate objects in the TOI.

To sum up, an abstract algebraic specification consists of a list of function names with the types of their arguments and result, and a set of axioms each of which say that two given terms have the same value. The axioms may include variables, in which case the axiom holds for all possible replacements of the variables by values of the appropriate type. The interested reader is directed to Ref. 2 for a more thorough treatment of this subject.

3. HOW TO CONSTRUCT AN ABSTRACT ALGEBRA

The idea of formal specification through the medium of abstract algebra and category theory has been around for some years and several specification languages based on this approach have appeared, notably Larch3 and Clear. Little has been said, however, about how a designer can go about constructing an abstract algebra that formalises his or her intuitive understanding of the thing to be specified.

This paper suggests a technique for constructing an abstract algebra to describe any given system which has state: it will be shown in action by two worked examples formally specifying the ‘back end’ to the (existing) QMC Message System. This is a real working message system in daily use by the Computer Science Department at Queen Mary College, and has suffered from well-intentioned but ill-fitting enhancements that have led to a number of strange inconsistencies in its behaviour. As such, the QMC mail system is far removed from the usual idealised systems considered in papers on formal specification.

3.1. Four stages

The technique to be described can be summarised as four steps:

- choose the types;
- choose the functions which change the state;
- choose the functions which display the state;
- choose a canonical form for the state.

These stages determine the majority of the detail involved in a specification, whilst clarifying the major aspects of the thing being described.

3.2. The system to be specified

The QMC Message System is an integrated message storage and message transmission system which enables users to send and receive arbitrarily large text messages. The recipients of messages can be either individual people, who have private mailboxes for this purpose, or public ‘bulletin boards’ which can be read by anyone. The bulletin boards are used for grouping messages relating to particular topics of discussion and the system maintains for each user a list of the bulletin boards to which he or she ‘subscribes’. By keeping track of the time at which each user last read the bulletin boards to which they subscribe and the time at which the most recent message was delivered to each bulletin board, the system can determine which boards contain messages which are new to the user. The system provides a way of stepping to the next board which contains new messages, without having to know its name, as well as the ability to examine any named bulletin board.

Entries on a bulletin board are shown as numbered single-line summaries, indicating author, date sent, title and size of message, with new items indicated by highlighting. To read the contents of a message, the user specifies its number and the READ command. The range of commands includes ways of searching for messages containing specific words, the ability to send messages in reply to items on a bulletin board or forward a message to another recipient, and the ability to delete unwanted messages.

All the above description of bulletin boards also applies to each user’s personal mailbox, but these are private and cannot be viewed by other users. The system interprets the name ‘mail’ to mean the user’s mailbox rather than a bulletin board called ‘mail’ and accessible to everyone.

4. THE MESSAGE DATABASE

This part of the system is concerned with storing ordered sequences of messages. Each message has an associated recipient name; this can be the name of a bulletin board* or the name of a user. It is the responsibility of the delivery program to check that the recipient name is currently valid; in the QMC Message System, faulty delivery programs can and do abandon messages to be stored even though no one can access them through the mail system!

4.1. Choosing the types

The persistent state of the message database will be represented by a single term. To allow successive operations which change the state of the system, these state-changing operations must be represented as functions from one state to another: for this reason there must be a type messagebase to represent the persistent state of the message database.

The other types of objects that need to be represented are names and messages, so types name and message are also required. No other types of object interest us at the moment. We will not need to say anything about the properties of these types, but we assume that it is possible to compare two names (for example) and say whether or

* Henceforth abbreviated to bb.
not they are the same. Technically, this ability is called an equality predicate and should be provided with the definition of the type, in this case name. Questions like ‘Is a name a string of characters?’ and ‘Are two strings of characters equal if their first 8 characters are identical?’ are completely irrelevant at this level of abstraction; all we need is the statement that names exist and there is an equality predicate on names.

We freely allow ourselves to use types such as sequence of X and set of X, confident that axioms for such things exist in libraries. They will be equipped with the usual functions on sequences and sets, with the extra operation \( \Diamond \) which appends an object of type X to an object of type sequence of X producing a new sequence.

We also allow the construction of disjoint unions, denoted by type1 | type2, which means a type whose objects are either from type1 or type2 and in which all ambiguity is removed by some (implicit) tagging of the object with its original type. This is most often used to add a distinguished object to an existing type, e.g. message | ERASED, which is the type of messages and a special distinguished object called ERASED which is not to be confused with any of the message objects.

4.2. Choosing the functions that change the state

These are the functions that the designer intuitively knows and so they must drive the specification. In the message database, the state-changing operations are:

- create_messagebase
- initialise the empty message database
- deliver_message
- add a message to the message database
- delete_messages
- remove all the messages for a given recipient

The provision of an operation to delete all messages for a given recipient is a design choice: other systems may have a facility to remove a single message but the original author of the Message System chose not to.* The above operations must all be functions which have result-type messagebase, so that we can construct terms from them to represent the possible states of the message database. Functions whose result type is the type we are specifying are called constructor functions (or just constructors) and the aim of Step 2 in the methodology is to identify all of the constructor functions to be specified. The full type specification of the chosen functions is:

- create_messagebase: \( \rightarrow \) messagebase
- deliver_message: 
  \( \text{messagebase} \times \text{name} \times \text{message} \rightarrow \text{messagebase} \)
- delete_messages:
  \( \text{messagebase} \times \text{name} \rightarrow \text{messagebase} \)

Notice that nothing has yet been said formally to describe these functions; they have merely been given suggestive names and some descriptive comments. The thing specified so far is identical to the following abstract algebra:

- \( F_1: \text{Types} \ S_1, S_2 \text{ and } S_3 \rightarrow S_1 \)
- \( F_2: S_1 \times S_2 \times S_3 \rightarrow S_1 \)
- \( F_3: S_1 \times S_2 \rightarrow S_1 \)

* Or at least, chose not to provide a function to delete single messages directly. See sections 6 and 8 below for details of how users clean out their mailboxes.

The statement of the properties of the functions is derived from the designer’s intentions in the next two steps. After completing them, there will be a formal, axiomatic statement of what each function does to the state of the message system.

There is one other decision which can be made before proceeding the next stage: the designer’s intuitive understanding of the system being specified will normally include knowledge of which functions are just ‘undoing’ the effects of other functions. If you had to reconstruct a given state, are there any functions which you would never need to use? In the case of the message database, the answer is Yes: any possible object of type messagebase can be constructed without using the delete_messages function. Since deleting all the messages for some recipient is the same as never sending any, we can just forget the delete_messages function and any preceding deliver_message functions which have the same recipient. This general ability to convert terms involving delete_messages into terms which don’t lead to a new definition; we call delete_messages a convertible constructor function.

Identifying the subset of convertible constructor functions has important consequences for the axioms in the specification. In particular, in defining what happens when a function is applied to any object of type messagebase, we don’t need to give axioms for messagebases whose outer function is one of the convertible constructor functions. The reason for this is that any such messagebase term can be converted into one which doesn’t involve these constructors.

4.3. Choosing the functions which display the state

There is little value in having a system which maintains persistent state information if there is no way of displaying that state. The set of observation functions that we choose for the message database consists of a single function which returns the sequence of messages held for a given recipient:

- \( \text{get_bb}: \text{messagebase} \times \text{name} \rightarrow \text{sequence of [message]} \)

This function does not have result type messagebase so it cannot appear as the outer operation of a term which represents the system state: it is therefore a function without side-effects, hence the name observation function. But what does it do? It is now necessary to state for every possible message database \( M \), what the result of applying get_bb will be: every database is built from the functions of the previous section, so a recursive definition with a case for each of these functions will suffice. We can give an axiom for the case get_bb(delete_message(M,x),y) or omit it as we please, because delete_message is a convertible constructor function. If we give an axiom, it must agree with the result of applying get_bb after converting the delete_messages term.

The axiomatic definitions in the style of abstract algebra. The definitions that have an ‘if-then-else’ right-hand side are called conditional axioms and involve
various technical niceties about the existence of equality predicates. Such arguments are usually clear-cut in practical finite applications such as computer systems.

Each axiom is the answer to a question: for example, ‘What is the result of applying get bb to a messagebase with outermost term deliver_message?’ For those preferring a procedural viewpoint, the messagebase can be thought of as a trace of the sequence of operations applied, the question then becomes ‘What is the result of applying get bb after doing a deliver_message operation on the system?’

The axioms for get bb alone would be sufficient to define the message database if the final algebra semantics of Kamin was being used, because two terms are defined to be equal unless the observation functions can show them to be different. For the initial algebra semantics of the ADJ group, two terms are different unless the axioms can prove them equal and the further refinements of the next section become necessary. Recalling that the aim of this methodology is to write down the designer’s intentions, we will opt in favour of saying more about the constructor functions.

4.4. Choosing a canonical form for the state

The idea of a canonical form is a useful one and has many applications in mathematics and computer science. In this instance, it is used to determine further axioms necessary to describe the thing the designer has in mind.

A canonical form for a term describing the state is an ideal description of that state: if you wished to reconstruct the current system state anew, what would it look like? Answering this question has the effect of establishing exactly which parts of the history of the system (i.e. the precise structure of the term) are relevant and which are not. The distinction is similar to that between bags and sequences; sequences differ from bags by insisting that the order of the entries matters. Another example is the difference between a bag and a set. To reconstruct a set you only need to add all of the elements to the empty set; to reconstruct a bag you also need to know how many of each element there are, but in neither case do you need to know the order in which elements were added.

The pertinent state of the message system database is clearly the messages which have not been deleted, stored in the same sequence in which they were added. The sequence itself only matters for the messages addressed to a given recipient; the order of arrival of messages for different recipients doesn’t matter. It will be convenient to use the trace viewpoint again, and write ‘•’ to mean ‘the result of applying all the previous operations in sequence’. This simply allows us to write terms vertically instead horizontally and so reduces the nesting of brackets. The canonical form of the term describing the message database is therefore:

\[
\text{create messagebase} \\
\text{deliver message}(\bullet, \text{name1}, t_{11}) \\
\ldots \\
\text{deliver message}(\bullet, \text{name1}, t_{1k}) \\
\text{deliver message}(\bullet, \text{name2}, t_{2k}) \\
\ldots \\
\text{deliver message}(\bullet, \text{nameN}, t_{NM})
\]

* Strictly speaking, the distinguishing set can contain compound functions composed of both constructors and observation functions, but the observation functions are the most important bit.

We must now add axioms which allow the constructor functions to be rearranged in such a way that this canonical form can be achieved: it is not possible to automate this process, so practice and some good examples should serve to guide the designer.

We will invent two pieces of shorthand notation which should serve to make things clearer:

- \(\text{COMMUTES}\) means that \(f_1(f_2(M,x,...,z),p,...,q) = f_2(f_1(M,p,...,q),x,...,z)\), and will be used whenever there is no possible ambiguity about what is commuting. In particular, the result types of \(f_1\) and \(f_2\) should be the same, and both functions should only have one parameter of that type.

- \(\text{NO CHANGE}\) means that the term does not have an equivalent. This rather odd-sounding idea is used for conditional axioms where functions only commute given certain conditions. This should become clear when it is used in the examples.

In practice, commutativity is often much too powerful a property, but it will suffice for this first example. Recalling that we needn’t write axioms for the case where the messagebase term has a convertible constructor as its most recent operation, we must specify for all the constructors (including the convertible ones) what they do to terms involving the non-convertible constructors; in this case, create_messagebase and deliver_message:

\[
\text{deliver message}(\text{create messagebase}, x, t) = \text{NO CHANGE} \\
\text{if } x = y \text{ then } \text{NO CHANGE} \\
\text{else } \text{COMMUTES}
\]

The second axiom allows the operation of delivering a message to commute with other similar operations, provided that they do not involve the same recipient: the condition preserves the order of messages within a bulletin board but allows the messages for other bulletin boards to be shuffled into place in the canonical form.

\[
\text{delete message}(\text{create messagebase}, x) = \text{create messagebase} \\
\text{if } x = y \text{ then } \text{delete message}(M, y) \\
\text{else } \text{COMMUTES}
\]

The axioms for delete_message must allow any term involving this function to be re-written as one that doesn’t, because we have decided that it is a convertible constructor function. We need axioms which say what happens when you apply delete_messages after any other constructor function, just as in the definition of get bb. The axioms given can be thought of as ‘pushing’ each operation of delete_message down through the term, removing any relevant deliver_message operations in its path, until it gets to the original create_messagebase and expires. The last axiom allows delete_messages to commute with irrelevant applications of deliver_message: nothing must stand in the way of its inexorable plunge to destruction!

The reader may be wondering if the axioms do not allow the construction of terms other than the canonical form; can’t the axioms be used ‘the other way round’ to introduce spurious delete_message operations, for example? The answer is ‘Yes, they can’, but it doesn’t matter. All that matters is that the equivalent formu-
lations of the same system state be provably equivalent using the axioms.

4.5. Cross-checking

The axioms given for the convertible constructor functions have to be checked against those which define the observation functions to make sure that both sides of each equation are indeed observably the same. This is a process which could be automated, but it is relatively easy to do by hand. The problem is one of consistency: we are defining the operations on type messagebase, not redefining the natural numbers or any of the other types, so we do not want axioms that allow us to prove that \(1 = 2\) or that \(\text{True} = \text{False}\). How can such a situation arise? The answer is that supplying axioms which relate terms in the constructor functions leaves us open to problems in applying the axioms which define the observation functions. If term \(T1\) can be proven equal to term \(T2\) using the axioms, but \(\text{get}_{\text{bb}}(T1,x) \neq \text{get}_{\text{bb}}(T2,x)\) then we have introduced an inconsistency to our axioms. However, we believe that the Guttag & Horning approach\(^9\) of refusing to allow axioms relating constructor functions is counter-productive when trying to state the designer's intentions. For example, to specify ‘set of \(X\)’ without saying ‘adding an item twice is the same as adding it only once’ makes consistency checking trivial, but fails to make explicit one of the fundamental properties of sets.

Any axiom which includes \textit{COMMUTES} should be matched by a corresponding \textit{COMMUTES} clause in the axioms for the inner constructor, for example

\[
\begin{align*}
fl(f2(S,x),y) &= \text{if } x = y \text{ then } f1(S,y) \\
\text{else } \text{ COMMUTES} \\
fl(f1(S,y),x) &= \text{if } x = y \text{ then } ?? \text{ else } \text{ COMMUTES}
\end{align*}
\]

The \textit{COMMUTES} part of the first axiom carries the additional implication of the second (incomplete) axiom. The only choice remaining to the designer is what to put in place of ?? to complete the second axiom.

5. A MORE COMPLEX EXAMPLE: THE ADMINISTRATIVE DATABASE

The message system performs several other functions besides the maintenance of named lists of messages. These include:

- identifying new messages;
- distinguishing mailboxes from bulletin boards;
- keeping subscription lists.

Each user has a personal mailbox, but the system also has public bulletin boards which can be read by anyone. By keeping track of when messages are sent and when each user reads messages, the system can work out when new messages are waiting to be read by a given user. Not all users are interested in all bulletin boards (there are 119 boards in the current system), so the system also maintains a list of which bulletin boards are of interest to each user.

The specification of this aspect of the system is more complex, but follows the same sequence.

5.1 Types and operations

The types are \textit{adminbase}, \textit{name}, \textit{user}, \textit{natural} and \textit{boolean}. We require that the type \textit{user} come equipped with a function \textit{username} which takes a user and produces a name, but we don't require this function to have any special properties like ‘All usernames have no more than 8 letters’.

\[
\begin{align*}
\textit{username} &: \rightarrow \textit{name} \\
\textit{create}_{\textit{adminbase}} &: \rightarrow \textit{adminbase} \\
\textit{add}_{\textit{bb}} &: \textit{adminbase} \times \textit{name} \rightarrow \textit{adminbase} \\
\textit{delete}_{\textit{bb}} &: \textit{adminbase} \times \textit{name} \rightarrow \textit{adminbase} \\
\textit{subscribe} &: \textit{adminbase} \times \textit{name} \times \textit{user} \rightarrow \textit{adminbase} \\
\textit{de}_{\textit{subscribe}} &: \textit{adminbase} \times \textit{name} \times \textit{user} \rightarrow \textit{adminbase} \\
\textit{have}_{\textit{read}} &: \textit{adminbase} \times \textit{name} \times \textit{user} \times \textit{natural} \rightarrow \textit{adminbase} \\
\textit{have}_{\textit{written}} &: \textit{adminbase} \times \textit{name} \times \textit{user} \times \textit{natural} \rightarrow \textit{adminbase} \\
\textit{bb}_{\textit{exists}} &: \rightarrow \textit{boolean} \\
\textit{bb}_{\textit{list}} &: \rightarrow \textit{set of}[\textit{name}] \\
\textit{subscription}_{\textit{list}} &: \rightarrow \textit{set of}[\textit{name}] \\
\textit{last}_{\textit{read}} &: \rightarrow \textit{natural} \mid \textit{NEVER} \\
\textit{last}_{\textit{write}} &: \rightarrow \textit{natural} \mid \textit{NEVER} \\
\end{align*}
\]

The convertible constructors are \textit{delete}_{\textit{bb}} and \textit{de}_{\textit{subscribe}}, which just ‘undo’ matching calls to \textit{add}_{\textit{bb}} and \textit{subscribe}.

5.2 Axioms for the observation functions

There are considerably more observation functions for this part of the mail system, so the axioms will be grouped together for clarity:

\[
\begin{align*}
\textit{bb}_{\textit{exists}}?(\textit{create}_{\textit{adminbase}},x) &= \text{FALSE} \\
\textit{bb}_{\textit{exists}}?(\textit{add}_{\textit{bb}}(A,y),x) &= \text{if } x = y \text{ then } \text{TRUE} \\
\textit{bb}_{\textit{exists}}?(A,x) &= \text{if } x = y \text{ then } \text{TRUE} \\
\textit{bb}_{\textit{exists}}?(\textit{subscribe}(A,y,u),x) &= \textit{bb}_{\textit{exists}}?(A,x) \\
\textit{bb}_{\textit{exists}}?(\textit{have}_{\textit{read}}(A,y,u,n),x) &= \textit{bb}_{\textit{exists}}?(A,x) \\
\textit{bb}_{\textit{exists}}?(\textit{have}_{\textit{written}}(A,y,u,n),x) &= \textit{bb}_{\textit{exists}}?(A,x) \\
\end{align*}
\]

The last three axioms for \textit{bb}_{\textit{exists}}? each mean ‘This constructor is not relevant to the observation function \textit{bb}_{\textit{exists}}?’ We cannot separate the administrative database into a subscriptions part and a bulletin boards part because the two are too closely interdependent; however, we can introduce another piece of notation to
both shorten the text of our specification* and emphasise the independence of the observation function and these constructors.

IGNORES. The statement ‘obf IGNORES cf1, ..., cfn’ is shorthand for axioms obf(cf1(x,...,z),p,...,q) = obf(A,p,...,q) for i ranging from 1 to n. It can only be used when obf is an observation function and when there is no ambiguity about what the axioms should be. In particular, the result types of the cf should all be the same, the observation function should have only one parameter of that type and each cf should have only one such parameter.

This allows us to sum up the last three axioms as

\[
\text{bb_exists} \quad \text{bb_add} \quad \text{bb_have_written} \\
\]

This shorthand may only be used for observation functions: if sieve and lump are constructors with the axiom sieve(lump(X)) = sieve(X) then it would give entirely the wrong impression to write ‘sieve IGNORES lump’.

\[
\text{bb_list IGNORES subscribe, subscribe_list} \\
\]

bb_list IGNORES have_read, subscribe, have_written

bb_list(create_adminbase) = ∅

bb_list(add_bb(A,x)) = bb_list(A) ∪ \{x\}

\[
\text{subscription_list IGNORES subscribe, subscribe_list} \\
\]

subscription_list(create_adminbase,u) = \{username(u)\}

subscription_list(subscribe(A,x,v),u) = if u = v & bb_exists?(A,x) then subscription_list(A,u) ∪ \{x\} else subscription_list(A,u)

\[
\text{last_read IGNORES subscribe, have_read, subscribe_list} \\
\]

last_read(create_adminbase,u) = NEVER

last_read(have_read(A,x,v,n),y,u) = if u = v & x = y then n else last_read(A,y,u)

\[
\text{last_write IGNORES subscribe, have_read, subscribe_list} \\
\]

last_write(create_adminbase,u) = NEVER

last_write(have_written(A,x,v,n),y,u) = if u = v & x = y then n else last_write(A,y,u)

\[
\text{BB or not BB? If there is an application of add_bb(\bullet,x), then all subsequent have_read, have_written, subscribe or de_subscribe operations referring to x are significant. If there is no application of add_bb, then it is the mailbox of any user for whom username(user) = x. Only the applications of have_read and have_written by that user are relevant.}
\]

The given English description is far from clear, but the purpose of this paper is to produce a precise statement of what is intended. The reader may judge for himself/herself how successfully the axiomatic formalism achieves this effect. Without further ado, let us try to produce the axioms that formalise this canonical form, easy ones first.

\[
\text{delete_bb(create_adminbase,x)} \\
\]

= create_adminbase

\[
\text{delete_bb(add_bb(A,y),x)} \\
\]

= if x = y then delete_bb(A,x) else COMMUTES

\[
\text{delete_bb(subscribe(A,y),x)} \\
\]

= if x = y then delete_bb(A,x) else COMMUTES

\[
\text{delete_bb(have_read(A,y,v,n),x)} \\
\]

= if x = y & y = username(u) then delete_bb(A,x) else COMMUTES

\[
\text{delete_bb(have_written(A,y,v,n),x)} \\
\]

= if x = y & y = username(u) then delete_bb(A,x) else COMMUTES

\[
\text{de_subscribe(create_adminbase,x,u)} \\
\]

= create_adminbase

\[
\text{de_subscribe(add_bb(A,y),x,u)} \\
\]

= COMMUTES

\[
\text{de_subscribe(subscribe(A,y),x,u)} \\
\]

= if x = y & u = v then de_subscribe(A,x,u) else COMMUTES

\[
\text{de_subscribe(have_read(A,y,v,n),x,u)} \\
\]

= if x = y & u = v then de_subscribe(A,x,u) else COMMUTES

\[
\text{de_subscribe(have_written(A,y,v,n),x,u)} \\
\]

= if x = y & u = v then de_subscribe(A,x,u) else COMMUTES

These are both examples of the familiar deletion process; delete_bb and de_subscribe both commute with or remove everything until they get down to create_adminbase and disappear. This is typical behaviour for operations which are convertible constructors.

\[
\text{subscribe(create_adminbase,x,u)} \\
\]

= create_adminbase

\[
\text{subscribe(add_bb(A,y),x,y)} \\
\]

= if x = y then NO_CHANGE else COMMUTES

\[
\text{subscribe(subscribe(A,y),x,u)} \\
\]

= if x = y & u = v then subscribe(A,x,u) else COMMUTES

\[
\text{subscribe(have_read(A,y,v,n),x,u)} \\
\]

= COMMUTES

\[
\text{subscribe(have_written(A,y,v,n),x,u)} \\
\]

= COMMUTES

The axioms to note are subscribe(add_bb(...) and subscribe(subscribe(...)); the then clause of the former is NO_CHANGE, which means that the two functions do not always commute, and that their sequence is important in certain cases. The latter axiom says that subscription

5.3 The canonical form

The canonical form for the Administration database is much more complex than that of the Message database. It can be handled most easily by describing it in various bits.

have_read & have_written. Only the most recent application of each of these functions is relevant for any given name/user combination. The numeric parameter is used by the user interface to determine which messages are new for each user.

subscribe. A subscribe(\bullet,x,u) operation is only relevant for bulletin boards. You can’t subscribe to non-bulletin boards (e.g. other users or spelling errors!).

* I am indebted to the referee for pointing out this way of condensing the specification.
is an idempotent operation, i.e. subscribing twice is the same as subscribing once.

The final operations, have_read and have_written, are both analogous to variable assignments; only the most recent application for a given (name, user) combination is remembered and all earlier applications are forgotten. The observation functions last_read and last_written provide the corresponding variable-like access facilities. Fortunately for us, have_read and have_written can move about fairly freely in the canonical form, but this is again typical behaviour for functions of this nature.

\[
\text{have_read}(\text{create_adminbase}, x, u, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = \text{username}(u) \text{ then} \\ \text{create_adminbase} & \text{else} \end{cases}
\]

\[
\text{have_read}(\text{add_bb}(A, x), y, u, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{have_read}(\text{subscribe}(A, x, y), y, v, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = y \& u = v \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{have_read}(\text{have_written}(A, x, u, n), y, v, m) = \begin{cases} \text{COMMUTES} & \text{if } x = y \& u = v \text{ then} \\ \text{have_written}(A, x, u, n) = \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{have_written}(\text{create_adminbase}, x, u, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = \text{username}(u) \text{ then} \\ \text{create_adminbase} & \text{else} \end{cases}
\]

\[
\text{have_written}(\text{add_bb}(A, x), y, u, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{have_written}(\text{subscribe}(A, x, y), y, v, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = y \& u = v \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{have_written}(\text{have_read}(A, x, u, n), y, v, m) = \begin{cases} \text{COMMUTES} & \text{if } x = y \& u = v \text{ then} \\ \text{have_read}(A, x, u, n) = \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{have_written}(\text{create_adminbase}, x, u, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = \text{username}(u) \text{ then} \\ \text{create_adminbase} & \text{else} \end{cases}
\]

\[
\text{have_written}(\text{add_bb}(A, x), y, u, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{have_written}(\text{subscribe}(A, x, y), y, v, n) = \begin{cases} \text{NO\_CHANGE} & \text{if } x = y \& u = v \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{have_written}(\text{have_read}(A, x, u, n), y, v, m) = \begin{cases} \text{COMMUTES} & \text{if } x = y \& u = v \text{ then} \\ \text{have_read}(A, x, u, n) = \text{COMMUTES} & \text{else} \end{cases}
\]

5.4 Use of observation functions in axioms

There is still one function to be specified, the add_bb constructor, and there are some problems about finding axioms for it. The COMMUTES parts of the axioms we have produced so far imply the following partial axioms:

\[
\text{add_bb}(\text{subscribe}(A, x, u), y) = \begin{cases} \text{?} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{add_bb}(\text{have_read}(A, y, u, n), x) = \begin{cases} \text{?} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

\[
\text{add_bb}(\text{have_written}(A, y, u, n), x) = \begin{cases} \text{?} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases}
\]

These do not help much in deciding what should happen in the \(x = y\) cases and it is here that we have to be careful.

Consider the following sequence of operations (where \(\text{username}(u) \neq x\)):

\[
\begin{align*}
\text{create_adminbase} \\
\text{have_read}(+, x, y, u, n) \\
\text{add_bb}(+, x) \\
\text{have_read}(+, x, y, m) \\
\text{add_bb}(+, x, y, m)
\end{align*}
\]

The intended canonical form for the state of the Administration database after this sequence of operations is:

\[
\begin{align*}
\text{create_adminbase} \\
\text{add_bb}(+, x) \\
\text{have_read}(+, x, y, m) \\
\text{have_read}(+, x, y, m)
\end{align*}
\]

In words, applications of have_read before \(x\) becomes a bulletin board are not recorded and applying add_bb\((+, x)\) has no effect if \(x\) is already a bulletin board. Allowing add_bb\((+, x)\) and have_read\((+, x, u)\) to commute is too powerful, because it would allow you to prove that all of the following states were identical:

\[
\begin{align*}
\text{create_adminbase} \\
\text{add_bb}(+, x) \\
\text{have_read}(+, x, y, m) \\
\text{have_read}(+, x, y, m)
\end{align*}
\]

(1) Desired form

\[
\begin{align*}
\text{create_adminbase} \\
\text{add_bb}(+, x) \\
\text{have_read}(+, x, y, m) \\
\text{have_read}(+, x, y, m)
\end{align*}
\]

(2) Too much

\[
\begin{align*}
\text{create_adminbase} \\
\text{add_bb}(+, x) \\
\text{have_read}(+, x, y, m)
\end{align*}
\]

(3) Too little!

The way to achieve this is to use the bb_exists? observation function to test for the existence of an outstanding add_bb operation without introducing the unwanted commutativity property into the axioms of add_bb. We introduce the following in place of ??? in each of the three axioms above.

\[
\text{add_bb}(A, x) = \text{if } \text{bb_exists?}(A, x) \text{ then } \text{A} \\
\text{else NO\_CHANGE}
\]

This allows us to make add_bb effectively idempotent, but without using commutativity to shuffle the application of add_bb down through the term to the previous application.

\[
\begin{align*}
\text{add_bb}(\text{subscribe}(A, x, u), y) = \begin{cases} \text{?} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases} \\
\text{if } \text{bb_exists?}(\text{subscribe}(A, x, y), y) \text{ then } \text{subscribe}(A, x, u) \\
\text{else NO_CHANGE}
\end{align*}
\]

\[
\begin{align*}
\text{add_bb}(\text{have_read}(A, y, u, n), x) = \begin{cases} \text{?} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases} \\
\text{if } \text{bb_exists?}(\text{have_read}(A, y, u, n), x) \text{ then } \text{have_read}(A, y, u, n) \\
\text{else NO\_CHANGE}
\end{align*}
\]

\[
\begin{align*}
\text{add_bb}(\text{have_written}(A, y, u, n), x) = \begin{cases} \text{?} & \text{if } x = y \text{ then} \\ \text{COMMUTES} & \text{else} \end{cases} \\
\text{if } \text{bb_exists?}(\text{have_written}(A, y, u, n), x) \text{ then } \text{have_written}(A, y, u, n) \\
\text{else NO\_CHANGE}
\end{align*}
\]
6. EXTENDING THE DEFINITION OF MESSAGEBASE

The algebraic specification for messagebase finished up as follows:

\[
\begin{align*}
\text{create_messagebase} & : \text{messagebase} \\
\text{deliver_message} & : \text{messagebase} \times \text{name} \times \text{message} \rightarrow \text{messagebase} \\
\text{delete_messages} & : \text{messagebase} \times \text{name} \rightarrow \text{messagebase} \\
\text{get_bb} & : \text{messagebase} \times \text{name} \rightarrow \text{sequence of \{message\}}
\end{align*}
\]

Now suppose we wish to add a constructor function erase_message which marks a message for removal by some suitable purging program, and an inverse operation unerase_message which removes the mark again. We choose to identify the message to be removed by its position in the sequence of messages generated by \text{get_bb} and we make two requirements:

1. erase_message(M, x, n) should have no effect if there are fewer than \( n \) messages for recipient \( x \).
2. The sequence returned by \text{get_bb} should contain a special non-message entry ERASED for messages which have been marked by erase_message.

This is quite hard but not impossible; it does however require a new trick. To say what we want, we allow ourselves to define extra 'hidden' operations which only exist for the purpose of specification. This is rather like using \text{bb_exists?} in the definition of \text{add_bb}, but the difference is that the functionality of the abstract type is not increased; no extra operations are added to the 'public' specification.

Handling requirement (1) is easy enough:

\[
\begin{align*}
\text{erase_message} & : \text{messagebase} \times \text{name} \times \text{integer} \rightarrow \text{messagebase} \\
\text{unerase_message} & : \text{messagebase} \times \text{name} \times \text{integer} \rightarrow \text{messagebase} \\
\text{erase_message}(M, x, n) & = \text{if length(get_bb(M, x))} < n \text{ then } M \text{ else } ???
\end{align*}
\]

The problems arise with requirement (2). Since \text{get_bb} gives us extra information, the canonical form must be extended to record successful applications of \text{erase_message} and \text{unerase_message}. Commutativity doesn't help because \( n \) is counting from the wrong end for simple 'counting-down' commuting, so we are forced to introduce a hidden constructor which we use in place of \text{erase_message} in the canonical form, and a hidden convertible constructor to use as a temporary function for locating the message we are erasing. For unerasing, we only need a hidden temporary operation.

\[
\begin{align*}
\text{erase_message}(M, x, n) & = \text{if length(get_bb(M, x))} < n \text{ then } M \text{ else erasing(M, x, length(get_bb(M, x)) - n)} \\
\text{unerase_message}(create_messagebase, x, n) & = \text{create_messagebase} \\
\text{unerase_message}(deliver_message(M, x, y), n) & = \text{if } x \neq y \text{ then COMMUTES} \text{ else if } n = 0 \text{ then erased_message}(M, x, t) \text{ else deliver_message(erasering(M, y, n - 1), x, t)} \\
\text{unerase_message}(create_messagebase, x, n) & = \text{create_messagebase} \\
\text{unerase_message}(deliver_message(M, x, y), n) & = \text{if } x \neq y \text{ then COMMUTES} \text{ else if } n = 0 \text{ then erased_message}(M, x, t) \text{ else erased_message(erasering(M, y, n - 1), x, t)}
\end{align*}
\]

The temporary function erasing gets around the problem of 'counting from the wrong end' by allowing us to adjust the numeric argument to erase_message; we subtract it from the length of the sequence generated by \text{get_bb} and then we are counting from the correct end. The axioms for erasing specify that it commutes with deliver_message and erased_message, subtracting one from the number each time it meets one of these functions for the stated recipient. When it meets such a message and its counter has dwindled to zero, it suddenly turns the deliver_message function into erase_message instead. Hidden functions are often used in this way, allowing the conversion of function parameters into a temporarily more convenient form for specification purposes.

It is important to recall that we are not trying to suggest a way of implementing these functions: we are simply describing their results. Any implementation of the 'visible' constructor and observation functions will do, provided the results it gives match those calculated from the axioms. The hidden functions can be thought of as analogous to local subroutines; the reader is invited to.
A FORMAL SPECIFICATION OF THE QMC MESSAGE SYSTEM

device alternative hidden functions to achieve the same effect.

The signature for messagebase ends up as shown below (the functions in the box are the hidden ones).

\[
\begin{align*}
create\_\text{messagebase} &: \rightarrow \text{messagebase} \\
deliver\_\text{message} &: \text{messagebase} \times \text{name} \times \text{message} \rightarrow \text{messagebase} \\
delete\_\text{messages} &: \text{messagebase} \times \text{name} \rightarrow \text{messagebase} \\
erase\_\text{message} &: \text{messagebase} \times \text{name} \times \text{integer} \rightarrow \text{messagebase} \\
erasure\_\text{message} &: \text{messagebase} \times \text{name} \times \text{integer} \rightarrow \text{messagebase} \\
get\_\text{bb} &: \text{messagebase} \times \text{name} \rightarrow \text{sequence of [message | ERASED]}
\end{align*}
\]

The axioms for get_bb, deliver_message and delete_messages must now be modified to take into account the new constructor function erased_message, and to fulfil requirement 2 above.

\[
\begin{align*}
deliver\_\text{message}(create\_\text{messagebase},x,t) &= \text{NO\_CHANGE} \\
deliver\_\text{message}(deliver\_\text{message}(M,x,t),y,s) &= \text{if } x = y \text{ then NO\_CHANGE } \\
&\quad \text{ else COMMUTES} \\
deliver\_\text{message}(erased\_\text{message}(M,x,t),y,s) &= \text{if } x = y \text{ then NO\_CHANGE } \\
&\quad \text{ else COMMUTES} \\
delete\_\text{messages}(create\_\text{messagebase},x) &= \text{create\_messagebase} \\
delete\_\text{messages}(deliver\_\text{message}(M,x,t),y) &= \text{if } x = y \text{ then delete\_messages}(m,y) \\
&\quad \text{ else COMMUTES} \\
delete\_\text{messages}(erased\_\text{message}(M,x,t),y) &= \text{if } x = y \text{ then delete\_messages}(m,y) \\
&\quad \text{ else COMMUTES} \\
get\_\text{bb}(create\_\text{messagebase},x) &= \text{nil} \\
get\_\text{bb}(deliver\_\text{message}(M,x,t),y) &= \text{if } x = y \text{ then get\_bb}(M,y) \circ f \\
&\quad \text{ else get\_bb}(M,y) \\
get\_\text{bb}(erased\_\text{message}(M,x,t),y) &= \text{if } x = y \text{ then get\_bb}(M,y) \circ \text{ERASED} \\
&\quad \text{ else get\_bb}(M,y)
\end{align*}
\]

7. COMBINING THE TWO DATABASES

The state of the QMC message system can now be represented as a pair of a messagebase and an adminbase. The operations, other than create_messagebase and create_adminbase, can all be extended to operations on the new type messagesystem so that they act on the appropriate element of the pair. The creation operations need to be linked together:

\[
\text{create} : \rightarrow \text{messagesystem} \\
\text{create} = \langle create\_\text{messagebase}, create\_\text{adminbase} \rangle
\]

The final specification is the axioms given in Sections 5 and 6 above, rewritten to act on objects of type messagesystem by modifying the appropriate component of the messagesystem pair. All of the messagebase constructors leave the adminbase component unaltered and commute with all the constructors derived from the adminbase specification. Likewise the adminbase constructors leave the messagebase component unchanged and commute with all the messagebase-derived constructors.

Only the public functions need to be converted because the hidden ones are only relevant to the messagebase or adminbase term in isolation; for example, the messagebase term can still contain the erased_message hidden constructor without having to have a new son_of_erased_message hidden constructor in the specification of messagesystem. The work of Goguen on the specification language Clear,4 and more recently LIL10 and OBJ2,1 is directed towards finding ways of combining abstract data types and covers the subject more deeply than we can in this paper.

8. THREE WAYS OF USING THE SPECIFICATION

Having produced an algebraic specification and hopefully clarified his or her ideas in the process, the designer can make further use of the specification without necessarily having to use it as an executable prototype.

8.1. Formal description of user interfaces

A user interface, by general consensus, enables a user to manipulate some application program(s). The formal algebraic specification described can stand for the programs that make up the 'application programs' part of the QMC Message System. The advantage of an algebraic formal model for this work is that the effects of the operations have been formally defined and so the effects of the user's commands can be calculated from the way they are built using these basic operations. The constructor and observation functions are the means by which the interface component of the system carries out the user's commands, and we have described precisely what they do without having to spend time saying how they do it. To illustrate this, we will describe the way the user-interface component can implement a 'purge' command.

One part of the QMC Message System allows the user to 'purge' erased messages from a named bulletin board or mailbox. The description of the user interface must describe not only how the user is able to persuade the system to perform this function, e.g. is there a PURGE command or menu option? how is the name selected?, etc., but also the meaning of the command in terms of its outputs and its effect on subsequent user-computer interaction. To handle the latter part of this description, we define the purge function in terms of the constructor and observation functions that we have specified.

\[
purge: \text{messagesystem} \times \text{name} \rightarrow \text{messagesystem} \\
purge(S,x) = \text{resend(S, get\_bb(S,x), delete\_messages(S,x))}
\]
concurrency. Thereby achieving better availability of the data and more database as a distributed collection of partial traces, and the quorum assignments can be tailored for each function, the problems of applying each function to a quorum as given by the axioms in the specification, the partial traces. Given knowledge of the properties of the database state is initially represented as a raw trace of the message and administrative databases, and several administrative file for each user, a configuration offering extensive concurrency of updates. Operations which cannot be concurrent, for example the delivery of several messages to a single bb, are made atomic and a locking scheme ensures that, in a conflict, one user is forced to wait (albeit for a very short time). Making one of two concurrent actions go first, followed by the other, is called serialisation in the concurrent systems literature.

The axiomatic specification given has no reference at all to concurrency. As far as we are concerned, the term representing the message system is accessible to all users, any of whom may apply constructor functions to it. All the public functions are atomic and get serialised in some unspecified way. The axioms, however, offer a lot of assistance to the implementor who wishes to allow concurrent access or to have multiple copies of the database in a distributed system. Weighl and Liskov point out that if two functions commute they can be serialised in either order without changing the net effect. This makes them candidates for concurrent execution if there is a way of interleaving the implementations of the operations; provided the two functions don’t interfere, no one minds in what order they finally complete their execution. An example in the message system would be two concurrent applications of deliver_message to different recipients: these operations commute and so, in principle, can be done at the same time. Note that concurrency and atomicity have nothing to do with the term-rewriting method of execution; the term-rewriting converts from one term into another one which means the same message_system object. This conversion merely makes life easier for the designer specifying the system and has no notion of ‘execution’ associated with it.

8.2. Exploiting concurrency

The QMC Message System allows concurrent access to the message and administrative databases, and several users can attempt to deliver messages simultaneously. The implementation uses a file for each bb and an administrative file for each user, a configuration offering extensive concurrency of updates. Operations which cannot be concurrent, for example the delivery of several messages to a single bb, are made atomic and a locking scheme ensures that, in a conflict, one user is forced to wait (albeit for a very short time). Making one of two concurrent actions go first, followed by the other, is called serialisation in the concurrent systems literature.

The axiomatic specification given has no reference at all to concurrency. As far as we are concerned, the term representing the message system is accessible to all users, any of whom may apply constructor functions to it. All the public functions are atomic and get serialised in some unspecified way. The axioms, however, offer a lot of assistance to the implementor who wishes to allow concurrent access or to have multiple copies of the database in a distributed system. Weighl and Liskov point out that if two functions commute they can be serialised in either order without changing the net effect. This makes them candidates for concurrent execution if there is a way of interleaving the implementations of the operations; provided the two functions don’t interfere, no one minds in what order they finally complete their execution. An example in the message system would be two concurrent applications of deliver_message to different recipients: these operations commute and so, in principle, can be done at the same time. Note that concurrency and atomicity have nothing to do with the term-rewriting method of execution; the term-rewriting converts from one term into another one which means the same message_system object. This conversion merely makes life easier for the designer specifying the system and has no notion of ‘execution’ associated with it.

8.3. Replication in distributed implementations

Work by Herlihy considers the representation of the database as a distributed collection of partial traces, and the problems of applying each function to a quorum of the partial traces. Given knowledge of the properties of that function as given by the axioms in the specification, the quorum assignments can be tailored for each function, thereby achieving better availability of the data and more concurrency.

The properties of interest are concerned with serial dependency of operations, i.e. the preceding operations that will affect the result of the operation to be applied. For example, the effect of add_bb(+,x) or subscribe(+,x) depends only on previous add_bb(+,x) and delete_bb(+,x) operations, whereas subscribe_list(+,y) depends on all three operations and delete_bb(+,x) on nothing at all. Constructor operations can never depend on preceding observation functions because the latter do not change the state.

Each operation is assigned a read quorum, indicating the number of copies of the data that need to be examined in order to be sure of seeing all the relevant preceding operations, and a write quorum which indicates the minimum number of copies of the database that must be updated by applying the operation. The read quorum for an operation f must ‘overlap’ the write quorum for each of the operations on which f is serially dependent. The delete_bb(+,x) read quorum is empty because the result doesn’t depend on any preceding calls, but its write quorum must be large enough to overlap with the read quorum of add_bb(+,x), amongst others.

Herlihy also considers the periodic compaction of these distributed traces by simplifying the traces. The ideal candidate for the compacted representation is the canonical form written down as part of the technique described in this paper.

9. RELATED ISSUES

9.1. Methodology

This work is closely related to that of Guttag and Horning, and our division of the signature into constructors, convertible constructors and observation functions parallels their distinction between CTERMS, ETERMS and OTERMS respectively. Both methods are justified by using structural induction over terms to show that axioms considering just the outer two functions of any term are sufficient to specify in full the behaviour of the abstract type. Our methodology departs from theirs, however, by allowing the designer to specify axioms relating the constructor functions; in the terminology of Ref. 9, this is giving axioms with CTERMS on both sides. The advantages of their restriction are that the resulting specification will be self-consistent and that the words algebra over the constructors is a suitable carrier for the abstract algebra. This must be offset against the disadvantage of being prevented from stating major properties of the type as axioms; as mentioned before, we feel that any specification of ‘set of X’ should clearly state that adding the same object to a set twice has the same result as adding the object only once. We are prepared to pay the additional cost of checking that the axioms for the observation functions respect the extra axioms we write, if only because this process also serves to verify the designer’s intuitive understanding of the observation function axioms.

Also closely related is the work of Furtado and Maibaum on the algebraic specification of databases. They describe a methodology based on traces in which the database state is initially represented as a raw trace of operations applied and subsequently represented by more refined traces, each derived from the raw trace, until a canonical form is produced. Each refinement stage has a corresponding set of procedures which show

322 THE COMPUTER JOURNAL, VOL. 31, NO. 4, 1988
how each constructor function modifies the trace, and how the observation functions are evaluated. An algebraic specification is obtained by converging the procedures of the final refinement into corresponding axioms. This process is intended to be accessible to the experienced database designer who may not have the mathematical background necessary to plunge straight into axiom writing, but we feel that the "How would I reconstruct this state?" approach is more direct, whilst equally illuminating.

9.2 Machine support for the methodology

This methodology offers ample opportunities for machine support in checking the input and prompting the designer for answers to all the necessary questions. A syntax-directed editor would assist in the construction of the axioms, checking the nesting and matching of brackets as well as making sure that the types of the expressions match the types of the functions applied to them.

More adventurous assistance could be given by software which presented the specification as a signature and an associated matrix of axioms. The matrix would have one row for each function and one column for each non-convertible constructor, the contents of cell at \((f_1, f_2)\) would be the axiom for \(f_1(f_2(\ldots), \ldots)\). This software could perform the checking necessary for uses of \textsc{commutes} and generate the implied partial axiom in the corresponding cell \((f_2, f_1)\).

The final aspect of machine support would be to provide a way of executing the observation functions and convertible constructors. This is feasible, since axiomatic specifications can be viewed as term-rewriting systems and hence as programs. The OBJ language of Goguen and Meseguer is an obvious candidate for this, but the axioms could also be translated into code in the ML or HOPE languages. More recent work by Goguen to extend OBJ by providing object-oriented programming constructs such as inheritance and local state offers a framework in which to combine separate objects which have state, whilst retaining the axiomatic style of specification and the term-rewriting model of execution. This work would assist steps such as those of Section 7, where the messagebase and adminbase states are combined into a single object.

The use of \textsc{commutes} and \textsc{no-change} offers valuable hints to a term-rewriting system, helping to prevent the looping problems often associated with commutativity. The division of the constructor functions into convertible and non-convertible is an additional hint; the evaluation of a convertible constructor should not halt until there are no convertible constructors in the term. Term-rewriting could also be used to check that the axioms for the constructor functions respect the axioms for the observation functions and do not introduce inconsistency. Scope exists for further research in these areas (the specification in this paper has been produced entirely by hand).

9.3 Notation

Amongst the various algebraic specification languages, the closest to the notation we have used is Larch, which has syntax for indicating hidden functions, for requiring types to be equipped with particular functions, and for indicating convertible constructors. Unfortunately, there is not yet any implementation of the Larch Shared Language which could be used to execute such a specification, so the next best notation is that of OBJ, which can be executed, but lacks the ability to indicate convertible constructors. It makes up for this by being executable and so allowing the designer to try out axioms quickly and fairly easily. It is also capable of using the \textsc{commutes} hints, but not the \textsc{no-change} ones; Larch has neither of these features.

To overcome the extreme nesting of brackets in complex terms, the higher-order function methods of functional programming can be used; for example, if we rearrange the order of the parameters of the \texttt{add_bb} function to get

\[
\text{add_bb}: \text{name } \times \text{messagebase } \rightarrow \text{messagebase}
\]

then we can consider \texttt{add_bb(x)} as a function

\[
\text{add_bb(x): messagebase } \rightarrow \text{messagebase}
\]

and then write the components of a term as a list, rather than nesting them, e.g.

\[
\text{add_bb(deliver_message(deliver_message(M,x,y),t),x)}
\]

becomes

\[
\text{add_bb(x) deliver_message(y,t) deliver_message(x,s) M}
\]

Furtado and Maibaum take this idea a step further by following a convention (used in some branches of mathematics) that the function application \(g(f(x))\) be written \[xfg\] and put the function applications in 'historical' order. This is similar to our occasional use of 

\[
\bullet \rightarrow \text{to refer to 'the term so far'.}
\]

9.4 Errors

None of the functions described in this specification returns any error conditions; it is quite in order to delete a bulletin board which does not exist. There are three reasons for taking this approach, only one of which is directly related to the axiomatic nature of the specification.

The first reason is simply that the specification describes an existing system which does appear to work this way. This may be related to the second reason, which is that it is much easier to exploit concurrency when implementing the specification if as many operations as possible are idempotent. If \texttt{delete_bb} returned an error indication when applied to an \texttt{adminbase} for which \texttt{bb_exists()} returns \texttt{FALSE}, then it would matter in what order two users attempted to delete a bulletin board. If we assume functions are applied to achieve their results, then we require merely that \texttt{delete_bb} return an \texttt{adminbase} for which \texttt{bb_exists()} returns \texttt{FALSE}. This allows applications of \texttt{delete_bb} to be serialised in any order, with all the attendant advantages for the implementor.

The third reason is a technical issue in algebraic specification: errors are hard to specify because all functions must be total. The best solution we know to this problem is that of OBJ, which puts error values onto a supersort and defines the function differently on that supersort, but we prefer to avoid such complications by arranging that, where possible, functions behave sensibly in all cases.

THE COMPUTER JOURNAL, VOL. 31, NO. 4, 1988 323
10. CONCLUSIONS

We have presented a method for getting ideas out of a designer's head and onto paper. This method produces a mathematical description which can, in principle, be executed by a suitable system, but which can also be examined for consistency, used as the basis for further formal descriptions or used as an aid to the implementor. The mathematics is derived fairly directly from the designer's understanding of the system being specified without requiring a detailed understanding of its theoretical basis. Given a stock of examples to draw on (the ones in this paper for example), it should be possible for any designer or programmer to produce an axiomatic specification using the method described, though without machine support it is difficult to manage even a modest-sized specification.

Acknowledgements

The author wishes to thank his colleagues at Queen Mary College who have contributed to this paper; Dr Peter Burton for teaching algebraic specification, Steve Cook, Jon Rowson and Stephen Sommerville for discussing early drafts, and Keith Clarke and Jean Dollimore for checking the axioms. The referee also made some suggestions which have been gratefully incorporated.

REFERENCES