National Physical Laboratory

NPL Report DITC 191/91
September 1991

LOTOS specification of the MAA standard, with an evaluation of LOTOS

Harold B Munster

ISSN 0262-5369

National Physical Laboratory
Teddington, Middlesex, UK, TW11 OLW

Approved on behalf of Director and Chief Executive, NPL,
by Dr T G Blaney, Head of Division of Information Technology and Computing

NPL Report DITC 191/91

September 1991

NATIONAL PHYSICAL LABORATORY

LOTOS specification of the MAA standard, with an evaluation of LOTOS

Harold B Munster

Abstract

This report explores the possible use of the formal language LOTOS to describe data security standards. It presents a LOTOS description of the MAA (message authenticator algorithm) standard, but identifies problems with using LOTOS for other data security standards.

Contents

Page

Summary

The data security group at NPL has studied a number of formal languages, to evaluate their possible use for describing international data security standards. A formal language is one with mathematically defined rules and meaning. It is hoped that this work will lead to more rigorous methods for testing implementations of the standards.

This report studies one of the formal languages, called LOTOS. LOTOS was designed for specifying communication standards, and some data security standards are concerned with communication. However, the main purpose of this report is to examine whether LOTOS can describe the key *non-communication* aspect of data security standards, namely mathematical operations on data. This is important because this aspect can always be expected to arise in data security. Even for standards not concerned with communication, if one could use the same formal language as for communication standards, this might support a consistent approach to testing.

This report presents and explains a LOTOS description of the MAA (message authenticator algorithm) standard. This standard is concerned exclusively with mathematical operations, and so this exercise establishes that LOTOS does have potential for describing these.

However, the MAA standard has characteristics which make it easier to describe than other standards. This report explains these characteristics, and indicates how data security standards without them create serious problems for LOTOS. This leads to the conclusion that LOTOS is unsuitable for describing most data security standards, although there may currently be nothing better when communication is involved. The problems stem from the method of defining data types in LOTOS, and may be addressed in a future version of the language.

This report assumes familiarity with basic programming concepts, but no knowledge of either LOTOS, the MAA standard, or special mathematical topics.

1 Overview of the problem: data security standards and formal languages

1.1 The nature of data security standards

This section gives a feel for the nature of standards which the data security group at NPL has begun to investigate. These are international (ISO) standards, particularly those related to *techniques* which achieve security, such as encipherment. Two important classes of standards are of interest: those related to algorithms, and those related to protocols.

The term "algorithm" has a modified meaning in data security. Its ordinary meaning is a method for achieving some result, eg sorting a list of numbers; this method specifies the detailed steps and the order in which they should be followed. In data security, one often performs transformations, such as enciphering a message, whose results are intuitively meaningless; this can mean that the easiest way to explain *what* the result is, is to specify *how* it is worked out. So a transformation (ie the overall task) is often defined using an (ordinary) algorithm, although it is the final result that matters rather than the route to it. This has led to the transformations themselves being called "algorithms". In this sense, if the same result can be achieved by a different method, eg by using look-up tables, this is not a different algorithm, but a different *implementation* of the algorithm.

Existing and prospective algorithm-related standards include those for message authentication, cipher operation modes, hash functions, and digital signatures.

A protocol gives procedures for interaction between separate entities. Protocol-related standards in data security are of two varieties: symbolic protocols, and OSI enhancements.

The symbolic protocol standards define procedures for achieving particular purposes, in general and abstract terms. Prospective standards include those for entity authentication and key management.

OSI (open systems interconnection) is a comprehensive system of standards for computer communication (not initially concerned with data security). These standards specify in detail the procedures for interaction, between various entities in a conceptual seven-layer structure, catering for all kinds of requirements and events. The OSI enhancements, referred to above, are extensions of these procedures which introduce the option of data security; this work began recently.

1.2 The MAA standard

This report studies the MAA (message authenticator algorithm) standard, ISO 8731 part 2. This is one of the algorithm-related standards mentioned in section 1.1. I explain, in this report, all the relevant material contained in the standard, according to its intended interpretation; for the actual text, see the standard itself [1].

The purpose of "authentication" here is to allow the receiver of a message to check that it is genuine. This is done by calculating a message authentication code (MAC) using a secret key, and sending it with the message. If any change is made to the message, the MAC should be different, but no-one can calculate the new MAC without knowing the secret key. The receiver knows the secret key, and uses this to determine the correct MAC; this will match the MAC received only if the message is the original one.

The purpose of the standard is to define the correct MAC value for any given message and secret key. In effect, it defines a *function*, in the mathematical sense, ie a relationship which connects every possible combination of message and key with a specific MAC value. This function is offered, by its publication in the standard, as one possible basis of a message authentication scheme; so any communicating parties can agree, if they so desire, to send, with their messages, the MAC values determined by this function and some choice of key. (I have chosen to regard this function as the essence of the standard; certain complicating issues are considered in section 3.1.)

The standard uses "32-bit unsigned integers" as the basic units of information; these are called *blocks* in this report. Of the "inputs" to the above function, the key consists of a pair of blocks denoted by J and K, while the message consists of a sequence of one or more blocks; the standard states that the MAA must not be used to authenticate a message with more than 1 000 000 blocks. The "output" from the function is the MAC, which is a single block.

The standard actually defines two functions (a lower-level and a higher-level one), each taking a key and "message", and yielding a "MAC". The lower-level function is called "the algorithm"; using the key (J and K), it acts on a sequence of blocks and produces a "MAC". The higher-level function is "the mode of operation" of that algorithm; this is the function described above, acting on the "actual" message and producing the "actual" MAC. The mode of operation works by dividing the message into groups of 256 blocks, which are called *segments* in this report, and applying "the algorithm" to each segment; the "MAC" for each segment is prefixed to the next segment (for calculation purposes), the last "MAC" being the "actual" MAC.

The lower-level function in turn is defined as a three-part calculation: the *prelude* pre-processes the key, the *main loop* is repeated for each message block, and the *coda* yields the "MAC" for the segment. The standard first defines a collection of "very" low-level functions, called "the functions used in the algorithm", and then defines the prelude, main loop and coda by giving the steps to be followed (an algorithm in the ordinary sense); these steps use the "very" low-level functions.

1.3 Using formal languages

The standards described above are written in English, referred to as a "natural language". Examination of the MAA standard has highlighted a variety of areas in which it is not as explicit as it should be, and sometimes is misleading. It is very difficult to devise a natural-language definition whose meaning is clear to all readers, especially without becoming long-winded, because individuals vary in their background and in their ways of viewing and expressing things.

A *formal* language is based on a mathematical theory. It involves a set of symbols, to be combined according to precise rules, with a rigorously defined meaning. A suitable formal language makes it possible to write a definition (eg of an algorithm) whose meaning cannot be misunderstood, provided the reader understands the formal language. The mathematical basis of the language also enables such a definition to be analysed, so that its consequences can be established and errors discovered.

The data security group at NPL has been investigating the possible use of formal languages to express data security standards. This work has been motivated by the hope that it will lead to more rigorous methods for testing implementations.

Two formal languages, VDM and Z, have been used at NPL to write descriptions of the MAA standard [2, 3]. Both languages are well-suited to defining mathematical functions of the kind found in the MAA standard, and, presumably, other data security standards in the algorithm-related class. However, they do not provide a "ready-made" framework for describing concurrent processes, which arise in protocol-related standards.[1]

A special class of formal languages has been developed for describing OSI standards, in which concurrent processes are the central feature. ISO has adopted the term FDT (formal description technique) for languages in this class. There are three FDTs recognized by ISO: Estelle, LOTOS and SDL. Of these, LOTOS allows more abstract, implementation-independent descriptions to be written, and may therefore be better for definitive purposes. It is also currently the only one with a completely formal basis. It is defined in an international standard [4].

"LOTOS" stands for "language of temporal ordering specification". The essence of a LOTOS specification is to define the behaviour of a system. It does this by defining the order, in time, of observable events (interactions between the system and its environment), usually involving choices and concurrency. But it can also define an algebra of data values, because the observable events may involve data. This suggests the possibility of describing data security algorithms as well as protocols.

There could be considerable benefits in expressing algorithms and protocols in the same language. One might expect a data security protocol to involve some algorithmic element; and a single language for different standards might support a consistent approach to testing. It is of interest, therefore, to examine whether LOTOS is suitable for this. That is one purpose of this report.

I have used LOTOS to write a third formal description of the MAA standard. This demonstrates that a fairly complex algorithm can be expressed in LOTOS, even though there is no element of "temporal ordering" or behaviour. However, the MAA standard has certain characteristics which make it especially suitable for formal specification, as discussed in chapter 5. The second purpose of this report is to present this MAA description, explaining the technical issues.

[1] This may not always be a serious problem. Initial studies of symbolic protocol standards in data security suggest that the interactions follow a fixed sequence; so processes are "concurrent" only in a limited sense, and concurrency features in the formal language may not be important.

2 A tutorial on the parts of LOTOS used in describing the MAA

2.1 Concepts involved in an abstract data type in LOTOS

2.1.1 Sorts and operations

A typical programming language provides several basic data types, such as "integer", as well as methods of building more complex data types out of basic ones. An example of a more complex data type is a "bit string", ie a string of one or more bits with no maximum length; this will be used to illustrate various concepts in this chapter.

To use bit strings in a programming language, one might have to design a mechanism for constructing them, eg using pointers. Then one might write short routines to manipulate bit strings, eg to append a new bit to an existing bit string. This is an example of a *concrete* data type, which involves implementation-specific features.

An *abstract* data type is concerned with the essential *principles* of objects like bit strings. For example, a bit string must be one of the distinct objects "0", "1", "00", "01" etc; but one is not concerned with any machine representation of these objects. Similarly, one is concerned with the relationship between the bit string "11010", the bit 1, and the bit string "110101" (namely, that "11010" with 1 appended gives "110101"); but not with how to accomplish this operation in practice. It is these abstract properties that matter as far as specification (rather than implementation) is concerned.

Two concepts have just been illustrated:

(a) There is the concept of a class of objects, such as the set of all possible bit strings. In programming languages, this is usually called a "type", but in LOTOS it is called a *sort*. (A "type" means something else in LOTOS.)

(b) There is the concept of an *operation*, such as appending a bit to a bit string. In a programming language, this involves an *action* using "physical" objects, but in LOTOS it is simply a *relationship* between objects (like "11010", 1 and "110101").

Each sort and operation in LOTOS is given an identifier. One may write, for example:

```
sorts Bit, BitString

opns
    Append: BitString, Bit -> BitString
```

This introduces two sorts, `Bit` and `BitString`, and one operation, `Append`. These are intended to be as follows:

(a) `Bit` consists of all possible bits; that is, `Bit` is the class containing the two objects 0 and 1.

(b) `BitString` consists of all possible bit strings.

(c) If `bs` denotes any possible bit string while `b` denotes any possible bit, then `Append(bs, b)` denotes the bit string consisting of `bs` with `b` appended to it.

(These intended meanings will be fulfilled only when further definitions are given.)

In the above definition of `Append`, the phrase

```
BitString, Bit -> BitString
```

is called the *functionality* of the operation. It means that this operation "accepts" two *arguments*, or "input" parameters, which belong to the sorts `BitString` and `Bit` respectively; and that it yields a single *result*, which belongs to the sort `BitString`. This explains the form of the expression `Append(bs, b)` above. Note, here, that `bs` and `b` can be any *expressions* denoting objects of the right sorts. Note, too, that `Append(bs, b)` itself is just another expression of sort `BitString` - an extra symbol to denote that particular bit string; it does not imply that an *action* occurs to produce that bit string from "physical" objects called `bs` and `b`. For example, if `bs` denotes "11010" while `b` denotes 1, then `Append(bs, b)` is simply one way of denoting "110101".

An operation in LOTOS can have any number of arguments, but must have exactly one result. It is always, in mathematical terms, a *total function*; this means it is a pairing, which connects every possible string of "input" objects (matching the argument sorts listed in the functionality) with a specific "output" object (matching the result sort). With `Append`, for example, `bs` and `b` can be *any* `BitString` object and *any* `Bit` object, respectively; this "input" combination always corresponds to a specific "output" object, `Append(bs, b)`, selected from the sort `BitString`. This contrasts with a *partial function*, where some "input" combinations may correspond to an "output" object, but other combinations may be invalid (even though matching the argument sorts).

The notation `Append(bs, b)` is like that used in many programming languages for a function, especially one written by the programmer; it is called "prefix" notation (operation *before* the arguments). An alternative format, eg `bs + b`, is also common in programming languages, especially for built-in operations; this is "binary infix" notation (operation *between* two arguments). LOTOS allows binary infix notation to be used for an operation, provided it has exactly two arguments. To use such a + operation instead of `Append`, one would write:

```
opns
    _+_: BitString, Bit -> BitString
```

Here, + is the identifier given to the operation (the rules for choosing identifiers are relatively flexible for operations). It is introduced between two underscores to indicate its binary infix usage.

The `Append` example shows that the arguments and result of an operation can involve a mixture of sorts, `BitString` and `Bit` in this case. In general, one builds a "package" containing several sorts, and uses these as the argument and result sorts for several operations. This whole package of sorts and operations is called a *type* in LOTOS. The mathematical structure so defined is called a *many-sorted algebra*.

2.1.2 Establishing data values

Each sort is a set of *values* - the complete set of distinct objects of that sort, eg 0 and 1 in the case of `Bit`.

When one introduces an operation such as

```
Append: BitString, Bit -> BitString
```

the effect is actually to *create* `BitString` values: for every value `bs` already present in the sort `BitString` and every value `b` present in the sort `Bit`, this operation *creates* another value, `Append(bs, b)`, and this is added to the range of values in `BitString`. (This value may not be strictly "created" by `Append`, because one may define it to be a value already existing in `BitString`.) Note that `bs` here may be a value itself created by `Append`.

Indeed, operations are the *only* means of creating values: a value exists only if some operation produces it for some combination of argument values. If `Append` is the only operation present, then there are actually no `BitString` values at all. This is because `Append` is the only available means of creating `BitString` values, but it, in turn, *needs* a `BitString` value as one of its arguments before it can produce any result. The intuitive interpretation, of course, is that `Append` should begin with a bit string, and use it to create a longer one; so it is clear that one should first provide the shortest bit strings, those with only one bit.

To create one-bit strings, one needs a second operation:

```
String: Bit -> BitString
```

For every `Bit` value `b`, `String` creates a `BitString` value, `String(b)`. This is intended to denote the bit string consisting of the single bit `b`.

If the only operations are `Append` and `String`, then there are still no `BitString` values. This is because both operations, to produce a result, need a `Bit` value as one argument, and there are no operations for creating `Bit` values. One therefore needs a third operation, whose result sort is `Bit`. Since one has not yet succeeded in creating values of any sort at all, there are no values which this third operation, in turn, can use as its arguments. To create the very "first" values, one must have *an operation with no arguments.*

Consider the definition:

```
1: -> Bit
```

This introduces an operation with the identifier `1`. The functionality states that `1` "accepts" *no* arguments, and yields a result of sort `Bit`. Recall, now, that the operation (a total function) connects every possible string of argument values with a specific result value. Here, there is only one possible string of argument values, namely *nothing*; and *this* corresponds to a specific result value. Recall that the result of `Append`, with arguments `bs` and `b`, is denoted by `Append(bs, b)` - the operation name, followed by two arguments enclosed in brackets. Likewise, the result of `1`, with *no* arguments, is denoted by `1` - the operation name, followed by no arguments and therefore, in LOTOS syntax, no enclosing brackets. The operation *creates* this value, an object of sort `Bit`. An operation with no arguments is called a *constant*, since its result is always the same; it is like a constant in a programming language - a fixed value.

With a similar constant `0`, the collection of sorts and operations becomes:

```
sorts Bit, BitString

opns
    0, 1:   -> Bit
    String: Bit -> BitString
    Append: BitString, Bit -> BitString
```

`0` and `1` can be introduced together because they have the same functionality.

Now consider what values are created by these four operations.

The operation `1`, as explained above, creates a value `1`, but since it has no arguments, it can never produce any other value; its use is exhausted. Likewise, the only value ever produced by `0` is `0`. These two values are of sort `Bit`. Since `0` and `1` are the only operations which produce `Bit` values, and both are exhausted, there cannot be any other `Bit` values. So the contents of `Bit` have been completely determined.

The argument to `String` is of sort `Bit`, and must therefore be `0` or `1`. Hence the result of `String` is either `String(0)` or `String(1)`; `String` *creates* these two values, which are of sort `BitString`. This exhausts the use of `String`. `BitString` now contains `String(0)` and `String(1)` (but nothing else so far). These represent the two strings consisting of one bit, "0" and "1".

With three operations exhausted, only `Append` can create further values. `Append` needs one `BitString` value and one `Bit` value as its arguments. So far, there are two possible values in each case, making four combinations, and so `Append` creates four corresponding results:

```
Append(String(0), 0)
Append(String(0), 1)
Append(String(1), 0)
Append(String(1), 1)
```

These are new values of sort `BitString`. They represent the bit strings "00", "01", "10" and "11", respectively, ie all possible strings of 2 bits.

This does not exhaust the use of Append: Append uses a BitString value as its first argument, and can therefore use the four new values. Since Bit has not changed, the second argument must still be 0 or 1. So the only *new* uses of Append combine one of the four new BitString values with 0 or 1, creating eight results:

```
Append(Append(String(0), 0), 0)
Append(Append(String(0), 0), 1)
Append(Append(String(0), 1), 0)
Append(Append(String(0), 1), 1)
Append(Append(String(1), 0), 0)
Append(Append(String(1), 0), 1)
Append(Append(String(1), 1), 0)
Append(Append(String(1), 1), 1)
```

These, in turn, belong to BitString, representing all possible strings of 3 bits. They form further options for the first argument to Append; each can be combined with 0 or 1, creating 16 *more* BitString values, which represent the strings of 4 bits. There is a cycle here: Append will next create all 5-bit strings from the 4-bit strings, then all 6-bit strings from the 5-bit strings, and so on.

Throughout this cycle, Append is the only operation still active. The key condition at each stage is that the only BitString values not already used represent all bit strings of some fixed length *n*. Therefore, the new uses of Append combine an *n*-bit string with 0 or 1, creating all (*n*+1)-bit strings. After this step, all the *n*-bit strings have been used, but the (*n*+1)-bit strings have not, and thus the key condition is met once again. The cycle continues "for ever", and so all bit strings of *any* length are created in due course; but it never produces anything *but* bit strings (ie values clearly *understood* to represent bit strings).

The outcome of this process is that BitString contains one value representing every possible bit string. So both Bit and BitString contain the intended values. This outcome follows from the list of sorts and operations: the functionalities of the four operations dictate the contents of the two sorts, in the way just explained.

This definition of bit strings is abstract. For example, while it establishes the existence of a value Append(String(0), 1), and while this *represents* the string "01", there is nothing to actually specify that Append(String(0), 1) denotes "01". The symbols "0", "1", "00", "01" etc are really a notation; the LOTOS definition introduces a different notation for the same objects, ie String(0), String(1), Append(String(0), 0), Append(String(0), 1) etc. Append(String(0), 1) *is the name* of a string; its intuitive meaning is clear. This name is made up of the successive operations used to create that string (0, 1, String and Append).

2.1.3 Equations

Consider a fifth operation:

```
Prefix: Bit, BitString -> BitString
```

If `b` denotes any bit while `bs` denotes any bit string, then `Prefix(b, bs)` is intended to denote the bit string consisting of `b` prefixed to `bs`. But `Prefix` actually *creates* a `BitString` value for every possible argument combination; for example, since `1` is a `Bit` value and `String(0)` is a `BitString` value, `Prefix` creates a new `BitString` value named `Prefix(1, String(0))`. Moreover, the values created by `Prefix` can be used as arguments to both `Append` and `Prefix`, creating further `BitString` values; and these, in turn, can be used as arguments to create yet more values, and so on.

This avalanche of new values is unwanted. The first four operations created the complete set of bit strings. When one introduces `Prefix`, one does not want it to create any *new* bit strings; instead, the result of `Prefix` is always intended to be one of the bit strings already existing. For example, `Prefix(1, String(0))` is not intended to be a new bit string, but is intended to denote the string "10", which already exists with the name `Append(String(1), 0)`.

One now wants *more than one expression to denote the same object.* Each individual expression, or "name" for an object, such as `Prefix(1, String(0))`, made up of operation identifiers, is called a *ground term.* The object denoted by that expression is a *value* of some sort. So one now wants to allow different ground terms to denote the same value.

For this purpose, LOTOS allows one to write *equations.* An example is:

```
Prefix(1, String(0)) = Append(String(1), 0);
```

This equation contains, on the left and right, two ground terms formed using the available operations, both of sort `BitString`. It means that these denote the same value (the same bit string). One may view this value as being created either by `Append` or by `Prefix`; if one views it as being created by `Append` (as in section 2.1.2), for example, then one may think of `Prefix` as "reaching it by an alternative route", or "creating it" (loosely speaking) "a second time". This is why I qualified the statement, early in section 2.1.2, that an operation *creates* its result values: some of the values "created" by operations can be "duplicates" of one another.

A more comprehensive form of equation is illustrated by:

```
forall x, y: Bit, s: BitString

Prefix(x, Append(s, y)) = Append(Prefix(x, s), y);
```

This uses *variables,* called `x`, `y` and `s`. The meaning is that `x` and `y` each can denote any `Bit` value, while `s` can denote any `BitString` value, and the equation always holds. The intuitive interpretation is that if one starts with any bit string `s`, appends any bit `y`, and then prefixes any bit `x`, one finishes with the same bit string as if one starts with `s`, prefixes `x`, and then appends `y`. The left and right sides are no longer ground terms (made up of operation identifiers only), but templates: in any *instance* of the equation, each variable could be replaced by a ground term denoting the same value as that variable, and the left and right sides would then become ground terms.

One may also write a *conditional equation*, eg:

```
    forall x, y: Bit, s, xs, sy: BitString

    xs = Prefix(x, s), sy = Append(s, y) => Prefix(x, sy) = Append(xs, y);
```

This means that the variables (`x`, `y`, `s`, `xs` and `sy`) can each denote any value of the right sort, and if the conditions on the left

```
    xs = Prefix(x, s)
    sy = Append(s, y)
```

are ever all met, then the equation on the right

```
    Prefix(x, sy) = Append(xs, y)
```

also holds. In practice, one can choose *any* values (of the right sorts) for `x`, `y` and `s`, and this fixes the sole values for `xs` and `sy` which meet the conditions; so the final equation holds for any `x`, `y` and `s`, with `xs` and `sy` as defined by the conditions. This example is equivalent to the previous one: in effect, it defines abbreviations, `xs` and `sy`, for intermediate expressions, and uses these to simplify the main equation.

In general, there will be a collection of equations accompanying the sorts and operations of a type. The equations form a vital part of the definition of the many-sorted algebra, since they affect the results of operations and the contents of sorts. How to devise an *effective* collection of equations, eg so that `Prefix` always yields the correct bit string, is shown in section 2.2.1.

2.1.4 Translating a data type into a many-sorted algebra

The *many-sorted algebra* is a conceptual mathematical structure, consisting of *sorts*, the *values* contained in those sorts, and *operations* relating those values. A *type*, or *data type*, can be regarded as a body of information which *determines* the many-sorted algebra: the *names* of the sorts, the *names* of the operations, the *functionalities* of the operations, and the *equations*. This section outlines how a data type determines a many-sorted algebra, according to the LOTOS semantics. This process underlies what is said in section 2.1.2 about the manner in which values are created.

First, the operation names and functionalities, together with the sort names, completely determine the set of possible *ground terms*, and the sort of each ground term. Section 2.1.2 examined the process of creating values, using the operations `0`, `1`, `String` and `Append`; this can be viewed, more precisely, as the process of generating the ground terms that denote those values. Introducing `Prefix`, as in section 2.1.3, generates additional ground terms on the same principles, even if they do not denote additional values. The principles are:

(a) If an operation has no arguments, then its name by itself forms a ground term. The sort of this term is the result sort of the operation.

(b) If an operation has one or more arguments, then for any list of ground terms already formed, whose sorts match the argument sorts of the operation, one can form a new ground term by enclosing that list in brackets and prefixing the operation name to it. The sort of this new term is the result sort of the operation. (The alternative notation, binary infix, is unimportant here; in fact, binary infix terms are formally reconstructed in a prefix format.)

In effect, each operation provides a *syntax rule* for forming ground terms of a particular sort. In (b), where a ground term is built from other ground terms, these in turn are formed by using the syntax rules recursively; in this way, all possible ground terms can be generated.

Next, the equations determine which ground terms are equivalent. If *A* and *B* represent ground terms, then the statement that *A* is equivalent to *B* (which may be true or false) is called an *assertion*. A *derivation* is a sequence of assertions, constructed in a way which ensures that these assertions are true. To be specific, the assertion that *A* is equivalent to *B* can be included in a derivation only if one of the following applies:

(a) One of the equations in the data type (or an instance of it, obtained by replacing each variable by a ground term of the right sort) states, without conditions, that *A* equals *B*.

(b) One of the equations in the data type (or an instance of it, obtained as in (a)) states, with one or more conditions, that *A* equals *B*; and all the conditions appear as earlier assertions in the derivation.

(c) *A* and *B* are the same ground term.

(d) An earlier assertion in the derivation states that *B* is equivalent to *A*.

(e) Earlier assertions in the derivation state that *A* is equivalent to *C*, and that *C* is equivalent to *B*.

(f) *A* consists of an operation name with one or more arguments; *B* consists of the same operation name with the same number of arguments; and earlier assertions in the derivation state that each argument in *A* is equivalent to the corresponding argument in *B*.

A derivation forms a proof of its final assertion. Rules (a) to (f) enable one to derive both direct and indirect consequences of the equations in the data type; these rules provide enough flexibility to prove *every* assertion one can infer intuitively from the equations. If it can be proved, using a derivation, that *A* is equivalent to *B*, then *A* and *B* are *interderivable*. If no such derivation is possible, then *A* and *B* are *not* interderivable; they can denote different values, in complete consistency with the equations.

Then, the complete set of ground terms can be split into classes, where two ground terms are put in the same class if they are interderivable, and in different classes if they are not interderivable. (There cannot be any conflicts, because of derivation rules (c), (d) and (e).) Thus the terms in a given class will all be equivalent, or, in the language of section 2.1.3, denote the same value; they will all have the same sort, because it is illegal to equate expressions of different sorts. Now, a *value* is formally defined as *one of these classes*; the values of any sort are those classes containing terms of that sort. A value is not formally given any "real" significance; it exists abstractly, and is identified by a collection of ground terms which denote it. In practice, a value will have some intuitive meaning, which may be obvious or may need informal explanation; and among the ground terms denoting it, there will typically be a "basic" one which is its natural representation.

The definition of a value as one of these classes of ground terms has an important consequence. If two ground terms are not interderivable, they are put in different classes, and therefore denote different values; that is, whenever ground terms *can* denote different values (the language used two paragraphs ago), they *do*. This is important because it establishes the range of distinct values in each sort. It is just as important to recognize differing values as to recognize identical values; but whereas one can write an equation to state that two expressions denote the same value, one does not write an inequality to state that two expressions denote different values; instead, different ground terms *always* denote different values unless otherwise stated (or implied). So the equations in the data type have a "hidden" meaning besides their obvious meaning: they are not only true, but, in a sense, the whole truth.

Finally, operations really act on values rather than ground terms. An operation was seen earlier as providing a syntax rule: for every list (empty in some cases) of "argument" ground terms, of the right sorts, it generates a "result" ground term. But the intention is that, for every list of argument *values*, of the right sorts, it should yield a result *value*. This presents no problem. Given a list of argument values, each value can be represented by any of the ground terms which denote it, and so the *list* of values can be represented by a *list* of ground terms, in one or more different ways; for each of these alternative lists of ground terms, the operation generates a different "result" ground term, but these "result" ground terms are all interderivable (because of derivation rule (f)), and so denote the same result *value*.

2.2 Defining data types in practice

2.2.1 Setting up a data type definition

This section develops a data type for bit strings, containing the two sorts and four operations of section 2.1.2, the operation `Prefix` of section 2.1.3, and an operation `Concatenate` which concatenates two bit strings.

To develop a data type, one begins with an *intuitive* idea of the sorts wanted. One gives each sort a name, and has in mind the range of values it should contain. For example, `Bit` should contain 0 and 1, while `BitString` should contain all strings of one or more bits.

The first task is to devise a minimum set of operations that create all the values wanted. These key operations, each needed to help build a sort, are called the *constructor* operations. This is not a formal designation: the constructors are simply the operations introduced first, which are enough to establish all the values. As shown in section 2.1.2, `0` and `1` serve to establish all `Bit` values, while `String` and `Append` then establish all `BitString` values. So these four operations are the constructors for `Bit` and `BitString`.

In general, there are three things to check:

(a) One must have an intuitive idea of what the constructors do. For each operation, one chooses its functionality. This enables one to consider what the argument values can be, *using the intuitive range of values one has in mind* for each sort. For any possible combination of argument values, one must satisfy oneself what the result is, within the intuitive range of values in mind for the result sort.

Thus, `0`, with no arguments, is meant to be the bit 0, while `1` is likewise meant to be the bit 1; `String(b)` is meant to be the bit string consisting of just `b`; and `Append(bs, b)` is meant to be the bit string consisting of `bs` with `b` appended to it.

(b) One must ensure that every value wanted can be obtained somehow using the chosen constructors.

The intended `Bit` values 0 and 1 can be obtained using the operations `0` and `1`. Then, any intended `BitString` value can be obtained by introducing the first bit using `String`, and successively appending any remaining bits using `Append`.

(c) One must check whether any of the intended values can be obtained in more than one way, using the chosen constructors. If so, equations will be needed.

The bit 0 can be obtained only by using the operation `0`, with no arguments, while the bit 1 can be obtained only by using the operation `1`, with no arguments. Any one-bit string can be obtained only by using `String`, with the required bit. Any string of more than one bit can be obtained only by using `Append`, with `bs` consisting of all required bits except the last, and `b` being the last bit required. So no equations are needed for these constructors; each intended value is denoted by only one ground term.

Sometimes one cannot avoid "duplicating" values, even with this minimum set of operations. For example, to create sets, one needs an operation which yields the empty set, and an operation which adds one element to an existing set; then any set can obtained by starting with the empty set and adding the required elements, one at a time; but these elements can be added in any order, and with any number of repetitions, leading to many ground terms for the same set.

In such cases, one must devise a set of equations to accompany the constructor operations. These equations must meet two requirements:

(a) The equations must be intuitively valid. That is, given the *intuitive* meaning of the expressions involved, the left and right sides of each equation must truly represent the same value (given any conditions stated).

(b) All ground terms which *intuitively* represent the same value must be actually interderivable. Typically, one needs to classify the possible representations of a generic object (eg a set of *n* elements), and systematically trace their equivalence through the equations chosen.

After these steps, one has a preliminary data type. In the corresponding many-sorted algebra, the sorts, and the values they contain, are "final": the above principles ensure that they match the original intuitive intentions. Likewise, the operations introduced so far are completely and correctly defined; that is, they always yield the right result. This is what was achieved in section 2.1.2.

It remains to add (to the many-sorted algebra) any further operations wanted. These will not create any new values, but will form extra relationships between the values already existing. They can be introduced one by one; in each case, one decides the name and functionality of the new operation, and adds one or more equations at the same time. These new equations must simply define the result of the new operation, for all possible argument combinations; this result must be one of the values created by the constructor operations.

The next operation is:

```
Prefix: Bit, BitString -> BitString
```

To define the result, it is necessary to know more about the second argument, the "input" bit string. If this consists of just one bit, it can be denoted by `String(x)`, for some bit `x`; and prefixing a bit, which can be denoted by `y`, is meant to yield the bit string consisting of `y` followed by `x`. The existing notation for this result is `Append(String(y), x)`; this enables the result of `Prefix` to be spelt out in an equation:

```
forall x, y: Bit

Prefix(y, String(x)) = Append(String(y), x);
```

If the "input" string consists of more than one bit, it can be denoted by `Append(s, x)`, for some bit string `s` and some bit `x`. In order to prefix `y`, one first takes `s` by itself and prefixes `y` to that; then the extra bit `x`, at the end of the original string, is kept at the end of the new string:

```
forall x, y: Bit, s: BitString

Prefix(y, Append(s, x)) = Append(Prefix(y, s), x);
```

This is a recursive definition: the result of `Prefix` is defined in terms of another `Prefix` result. This is acceptable because the second `Prefix` expression is simpler: the "input" bit string is shorter. Applying the definition repeatedly will eventually lead to a one-bit "input" string, whereupon the result is defined *without* another `Prefix`.

The last operation is:

```
Concatenate: BitString, BitString -> BitString
```

As it happens, one need only consider the same two possibilities for the *second* argument. The first argument can be simply a bit string denoted by `t`:

```
forall x: Bit, s, t: BitString

Concatenate(t, String(x))    = Append(t, x);
Concatenate(t, Append(s, x)) = Append(Concatenate(t, s), x);
```

These two equations work in a similar way to those for `Prefix`.

The complete material for the data type must be arranged in the correct format. The whole type is given an identifier. The example type is called `BitString`, which is also the name of one of the sorts; type, sort and operation names are used in different contexts, and so the same identifier can be used for more than one of them without confusion.

```
    type BitString is

        sorts Bit, BitString

        opns

            0, 1:                    -> Bit
            String:                  Bit -> BitString
            Append:                  BitString, Bit -> BitString
            Prefix:                  Bit, BitString -> BitString
            Concatenate:             BitString, BitString -> BitString

        eqns

            forall x, y: Bit, s, t: BitString

            ofsort BitString

                Prefix(y, String(x))     = Append(String(y), x);
                Prefix(y, Append(s, x)) = Append(Prefix(y, s), x);

                Concatenate(t, String(x))     = Append(t, x);
                Concatenate(t, Append(s, x)) = Append(Concatenate(t, s), x);

    endtype
```

Sorts, operations and equations must be given in that order. The line

```
    ofsort BitString
```

states that the left and right sides of each subsequent equation are of sort `BitString`. There can be any number of "`ofsort`" statements, each introducing a group of equations. Variables defined before the first "`ofsort`" statement, like those here, are available to all the equations that follow. It is also possible to define variables immediately *after* an "`ofsort`" statement; these are available only to the equations in that group.

2.2.2 Importing simpler types into more complex ones

It is often useful to build a many-sorted algebra in stages. For example, imagine a system of data structures in which bit strings form just one component; for this, one could use the sorts `Bit` and `BitString`, with the operations defined in section 2.2.1, together with other sorts and operations. Yet `Bit` and `BitString`, with their own operations, by themselves form a natural collection of ideas, with a wider range of uses; it would therefore be helpful to introduce them in a separate *module*, and then include this in a larger system.

LOTOS therefore allows a data type to *import* the contents of one or more other data types. In effect, the new type then contains all the sort names, operation names, functionalities, and equations belonging to the imported types, besides any further sorts etc defined in the new type.

As an illustration, although somewhat meagre, the type `BitString` of section 2.2.1 could be developed in three stages: one to establish bits; one to establish bit strings, with a minimum of operations; and one to provide further operations. These are defined by three separate types, each importing the one before, where the last one is equivalent to `BitString` in section 2.2.1:

```
type Bit is

    sorts Bit

    opns

        0, 1:                   -> Bit

endtype


type BasicBitString is

    Bit

    sorts BitString

    opns

        String:                 Bit -> BitString
        Append:                 BitString, Bit -> BitString

endtype


type RicherBitString is

    BasicBitString

    opns

        Prefix:                 Bit, BitString -> BitString
        Concatenate:            BitString, BitString -> BitString

    eqns

        forall x, y: Bit, s, t: BitString

        ofsort BitString

            Prefix(y, String(x))    = Append(String(y), x);
            Prefix(y, Append(s, x)) = Append(Prefix(y, s), x);

            Concatenate(t, String(x))    = Append(t, x);
            Concatenate(t, Append(s, x)) = Append(Concatenate(t, s), x);

endtype
```

Note the format of each type definition: after the "`type...is`" heading, any *imported* types are first listed by name, and then any *new* sorts, operations or equations are given.

In this example, the three data types define three different many-sorted algebras. The third algebra is identical to that of type `BitString` in section 2.2.1, since `RicherBitString` includes the material of `Bit` and `BasicBitString`, and thus contains the same information as `BitString`. The first two algebras consist of *part* of the final one, with some sorts and operations stripped away.

A *well-written* collection of type definitions will normally meet the following requirements. In each intermediate many-sorted algebra, those sorts introduced so far should already be completely constructed. That is, any type which imports them should not alter the range of values contained in those sorts; it should only add completely new sorts or define new operations. In practice, this means two things:

(a) There should be no equation which relates *two results of imported operations*. Such an equation would either be redundant (because the two results are already equal), or make equal two values which are not equal in the earlier many-sorted algebra; in the latter case, it would reduce the range of values in one of the imported sorts.

All new equations should be concerned with defining the result of a *new* operation.

(b) Whenever a new operation uses an imported sort for its result, the result should be defined as an existing value: for every possible combination of argument values, the new equations should define the result as *one and only one* of the values previously existing. If they do not define it as any existing value, the result becomes a new value, and the imported sort is extended; while if they define it (directly or indirectly) as more than one existing value, those values become interderivable (all equal to the new result, and hence equal to one another), and the imported sort is reduced.

If these principles are not followed, some sort will be established with a certain apparent range of values, and then, on being imported to another type, will emerge with a modified range of values. This will form an obscure definition; if an operation used that sort for an argument, then the range of possible argument values, and hence the range of results, will be misunderstood.

2.3 Parameterized data types

One can work with other kinds of strings besides bit strings, eg character strings. Concepts like appending and concatenating apply to *all* kinds of strings, and so there would be many similarities between the data types defining these strings. To save duplicating such aspects, it is possible to define a *parameterized* data type, which defines strings of *any* kind. This can be "invoked", or *actualized*, as often as needed, to define strings of a specific kind.

The *parameters* of a type are either sorts or operations.

A sort parameter represents a set of values which are not specified. In the case of strings, the unspecified values are the individual elements from which a string is built. These elements will be bits in the case of a bit string, characters in the case of a character string, etc; but, for a string in general, they can be anything. For any given kind of string, there will be a sort consisting of all possible elements. For bit strings, this could be `Bit`; `Bit` is an example of an *actual parameter*, ie the *actual* sort used in a specific case. But within the "string" type, this sort may be called `Element`, since it can contain any kind of elements; `Element` is an example of a *formal parameter*, ie a name used to represent *any* sort.

An operation parameter is an operation whose argument and result sorts are all parameters. That is, it represents some unspecified operation, relating the unspecified values contained in the sort parameters. Again, the parameterized type uses a *formal parameter* to name the operation, while an *actual parameter* will be specified when a specific instance is defined. An example with operation parameters is described in section 2.4 (type `NonEmptyString`).

The formal parameters are called *formal sorts* and *formal operations*. It is also possible to write *formal equations*. Unlike ordinary equations, these are not used in determining a many-sorted algebra. Instead, they are *requirements*: they must already hold, once the *actual* parameters are specified, by virtue of the equations associated with those actual parameters. In the parameterized type definition, formal sorts, formal operations and formal equations must appear in that order, *after* any imported types, and *before* any non-formal sorts, operations and equations.

In essence, a parameterized type provides *part* of the information needed to determine the many-sorted algebra, namely that part which is common to different instances of the type. For example, a parameterized "string" type provides the sorts, operations and equations which *all* kinds of string have in common:

```
type String is

    formalsorts Element

    sorts String

    opns

        String:                  Element -> String
        Append:                  String, Element -> String
        Prefix:                  Element, String -> String
        Concatenate:             String, String -> String

    eqns

        forall x, y: Element, s, t: String

        ofsort String

            Prefix(y, String(x))     = Append(String(y), x);
            Prefix(y, Append(s, x))  = Append(Prefix(y, s), x);

            Concatenate(t, String(x))     = Append(t, x);
            Concatenate(t, Append(s, x))  = Append(Concatenate(t, s), x);

endtype
```

Compare this type definition with type `BitString` in section 2.2.1. The important difference is that "elements" are now unspecified, rather than being bits; therefore no constructor operations are given for the sort `Element`. When the actual elements come to be specified, the sort `String` will contain all possible strings of such elements, just as the sort `BitString` contained all possible strings of bits; while the operations `String`, `Append`, `Prefix` and `Concatenate` will correspond to those that had the same names before, performing analogous tasks but using the chosen elements rather than bits.

Strings of a specific kind can be defined by *actualizing* the type `String`. Actualization does two things:

(a) Any of the sorts and operations (including formal sorts and operations) of the parameterized type can be *renamed*, to reflect the specific instance being defined.

(b) One or more types are *imported*, which must, between them, contain all the *actual parameters*. That is, for every formal sort, one of the imported types must contain an actual sort with the right name (taking account of renaming); and for every formal operation, one of the imported types must contain an actual operation with the right name and functionality (again, taking account of renaming).

The effect of importing the types in (b) is to provide the rest of the information needed to determine the many-sorted algebra, eg constructor operations and equations for the actual parameters.

For example, this is how type `String` can be used to define bit strings:

```
type Bit is

    sorts Bit

    opns

        0, 1:                  -> Bit

endtype



type BitString is

    String actualizedby Bit using

        sortnames
            Bit for Element
            BitString for String

endtype
```

This defines two types, `Bit` and `BitString`. The first type appeared in section 2.2.2; it simply defines the sort `Bit`, with its constructor operations `0` and `1`. In the second type, the phrase

```
String actualizedby Bit
```

means that this type is an instance of type `String`, where type `Bit` is imported to provide the actual parameters. It is stated next that the type uses the new sort names listed, ie `Bit` and `BitString` instead of `Element` and `String`. New operation names can be listed in the same way (*after* any sort names), under the word "`opnnames`". Since the only formal parameter, `Element`, is renamed `Bit`, the *actual* parameter is the sort `Bit` imported from type `Bit`.

The effect of actualization is to create a *new* data type, like `BitString` above. The new type combines two lots of information:

(a) all the sorts, operations and equations of the *imported* types (like `Bit`);

(b) the renamed version of the *non-formal* sorts, operations and equations of the parameterized type that has been actualized (like `String`).

The *formal* sorts, operations and equations are superseded by the imported material, which includes the *actual* sorts and operations. By examining the contents of type `Bit`, and the non-formal part of type `String`, one can see that type `BitString` acquires the same sorts, operations and equations as type `BitString` in section 2.2.1; so it defines the same many-sorted algebra.

A parameterized type is not an "abstract data type" of the kind considered in section 2.1. It cannot be translated into any meaningful many-sorted algebra, because it contains only part of the information needed. It is simply a tool for building on other data types; the formal part places some requirements on those other types, while the non-formal part supplements their contents. It is the new type created by actualization that is a true "abstract data type" (assuming the imported types are not themselves parameterized).

2.4 Standard library types

LOTOS defines a *standard library* of data types, consisting of definitions of commonly needed types. One or more of these can be used in a LOTOS specification by writing a *library declaration.* For example, the library contains two types called `NaturalNumber` and `String`; the declaration

```
library

    NaturalNumber, String

endlib
```

makes these available, as if their definitions were written directly in the specification text.

This section covers the standard library types used in describing the MAA. It explains the sorts defined by these types, in terms of the values they contain and what these represent. It also explains those operations used in the MAA description, indicating what result they yield. To find out what equations are used for these definitions, what other operations are defined by these types, and what other types are available, see Annex A of [4].

(a) Type `Boolean`

The sort `Bool` consists of the values "true" and "false", generated by constructor operations:

```
    true, false: -> Bool
```

Other operations include:

```
    not:          Bool -> Bool
    _and_, _or_: Bool, Bool -> Bool
```

`not(x)`, `x and y`, and `x or y` give the conventional results of logical NOT, AND and OR functions.

(b) Type `NaturalNumber`

The contents of type `Boolean` are imported.

The sort `Nat` consists of the natural numbers, in the inclusive sense of 0, 1, 2 etc, generated by constructor operations:

```
    0:    -> Nat
    Succ: Nat -> Nat
```

`Succ(n)` represents the next number after `n`. The natural numbers are thus denoted by `0`, `Succ(0)`, `Succ(Succ(0))` etc. Other operations include:

```
    _+_, _*_:          Nat, Nat -> Nat
    _eq_, _le_, _gt_: Nat, Nat -> Bool
```

`m + n` and `m * n` give the conventional results of addition and multiplication. `m eq n`, `m le n` and `m gt n` give the status of the relations "`m` is equal to `n`", "`m` is less than or equal to `n`" and "`m` is greater than `n`".

(c) Type `NonEmptyString`

This is a parameterized type; it corresponds to type `String` in section 2.3, with some notational differences and extra operations.

The contents of types `Boolean` and `NaturalNumber` are imported. This provides a body of non-formal sorts, operations and equations, which will be carried into every actualized version of the type.

The main parameter is a formal sort called `Element`. This will consist of a range of "elements", the basic units making up strings.

Two formal operations called `eq` and `ne` are also used. If `x` and `y` are two "elements", `x eq y` will be `true` if `x` is equal to `y`, and `false` otherwise; while `x ne y` will be `true` if `x` is *not* equal to `y`, and `false` otherwise. The reason why these are *formal* operations is as follows.

Suppose the "elements" are actually bits. Then the *actual* sort may be called `Bit`, and contain the values `0` and `1`, as in section 2.3. `eq` and `ne` can yield a result of sort `Bool`, from type `Boolean`:

```
_eq_, _ne_: Bit, Bit -> Bool
```

Each of the arguments to `eq` is either `0` or `1`, and the result can be defined in all possible cases by the equations:

```
0 eq 0 = true;
0 eq 1 = false;
1 eq 0 = false;
1 eq 1 = true;
```

Since `ne` represents the opposite condition to `eq`, its result can be defined in all possible cases as follows:

```
forall x, y: Bit

x ne y = not(x eq y);
```

These equations define `eq` and `ne` for bits, but *different* equations are needed for other kinds of "element"; the parameterized type `NonEmptyString` cannot contain equations defining `eq` and `ne`, since the "elements" are unspecified. The imported type which actualizes `NonEmptyString`, providing the *actual* element sort, must provide *actual* equality and inequality operations, with equations written specially for those particular "elements". Thus `eq` and `ne`, along with `Element`, are *parameters*, remaining to be specified. `NonEmptyString` uses them to define other operations.

These formal operations introduce a technical complication. The argument and result sorts of a formal operation must themselves be formal. The *formal* operations `eq` and `ne` cannot use result sort `Bool`; they merely represent *actual* operations that must be imported from some other type, and those actual operations will yield a result of *some sort in that imported type*. `NonEmptyString` is not allowed to foretell what that actual result sort will be. It therefore uses another formal sort called `FBool`; but in practice the actual sort is always meant to be `Bool`, from type `Boolean`.

So `NonEmptyString` uses:

```
formalopns
    _eq_, _ne_: Element, Element -> FBool
```

This solution leads to a further complication. While the result sort of `eq` and `ne` cannot be specified as `Bool`, `NonEmptyString` still needs to use the associated operations `true` and `not`. These are therefore introduced as operations associated with `FBool`:

```
true: -> FBool
not:  FBool -> FBool
```

Since these are not new operations, but *represent* those associated with Bool, they are *formal* operations. They should not be renamed when NonEmptyString is actualized, but should represent the actual true and not of type Boolean.

So there are six parameters in all: Element, eq, ne, FBool, true and not.

The sort NonEmptyString will consist of all strings of one or more "elements", generated by constructor operations:

```
String: Element -> NonEmptyString
_+_:    Element, NonEmptyString -> NonEmptyString
```

String(x) will represent the string consisting of just x, while x + s will represent the string consisting of x prefixed to s. (In section 2.3, Prefix could be the second constructor instead of Append, because there is a symmetry between them.) Other operations include:

```
_+_:          NonEmptyString, Element -> NonEmptyString
_++_:         NonEmptyString, NonEmptyString -> NonEmptyString
Length:       NonEmptyString -> Nat
_eq_, _ne_: NonEmptyString, NonEmptyString -> Bool
```

s + x will represent the string consisting of s with x appended. s ++ t will represent the concatenation of s and t in that order. Length(s) will give the number of "elements" in s. s eq t and s ne t will give the status of the relations "s is equal to t" (ie the same string) and "s is not equal to t".

Notice that the names +, eq, ne, true and not have each been used for more than one operation. The operations using each name are distinguished by their functionalities. This is called "overloading operators".

(d) Type Bit

The contents of types Boolean and NaturalNumber are imported.

The sort Bit consists of the values 0 and 1, generated by constructor operations:

```
0, 1: -> Bit
```

Other operations include:

```
NatNum: Bit -> Nat
```

NatNum(b) gives the numeric "value" conventionally represented by b, ie 0 or Succ(0).

(e) Type Octet

The contents of type Bit are imported.

The sort Octet consists of the 256 possible octets, ie 8-bit codes or strings, generated by a constructor operation:

```
Octet: Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit -> Octet
```

Other operations include:

```
_eq_: Octet, Octet -> Bool
```

x eq y gives the status of the relation "x is equal to y". (Note that the contents of Boolean are imported via type Bit.)

(f) Type BitNatRepr

The contents of types Boolean, NaturalNumber and Bit are imported.

The sort BitString consists of all strings of one or more bits, generated by constructor operations:

```
Bit: Bit -> BitString
_+_: Bit, BitString -> BitString
```

Bit(x) represents the string consisting of just x, while x + s represents the string consisting of x prefixed to s. Other operations include:

```
_+_:         BitString, Bit -> BitString
_++_:        BitString, BitString -> BitString
_eq_, _ne_:  BitString, BitString -> Bool
NatNum:      BitString -> Nat
```

s + x represents the string consisting of s with x appended. s ++ t represents the concatenation of s and t in that order (ie s "prefixed" to t, or t "appended" to s). s eq t and s ne t give the status of the relations "s is equal to t" (ie the same string) and "s is not equal to t". NatNum(s) gives the numeric "value" conventionally represented by s, treating s as a binary numeral. Note that the *first* bit in the string, ie the bit which is *prefixed* to the rest of the string, is the *most significant* bit. For example, 1 + Bit(0) represents the string "10", which is the binary numeral for 2; so NatNum(1 + Bit(0)) is equal to Succ(Succ(0)).

(g) Type DecNatRepr

The contents of types Boolean and NaturalNumber are imported.

The sort DecDigit consists of the ten decimal digits, generated by constructor operations:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9: -> DecDigit
```

The sort `DecString` consists of all strings of one or more decimal digits, generated by constructor operations:

```
Dec: DecDigit -> DecString
_+_: DecDigit, DecString -> DecString
```

`Dec(x)` represents the string consisting of just `x`, while `x + s` represents the string consisting of `x` prefixed to `s`. Other operations include:

```
_+_:    DecString, DecDigit -> DecString
NatNum: DecString -> Nat
```

`s + x` represents the string consisting of `s` with `x` appended. `NatNum(s)` gives the numeric "value" conventionally represented by `s`, treating `s` as a decimal numeral. As with a bit string, the *first* digit is the *most significant* digit. For example, the following expressions represent the same string "1024", with numeric value 1024:

```
1 + (0 + (2 + Dec(4)))
Dec(1) + 0 + 2 + 4
```

The first expression uses constructor operations only, prefixing digits, while the second *appends* digits. The second expression uses fewer brackets, because binary infix operations are evaluated from left to right.

3 Describing the MAA - general observations

3.1 What is being defined

(a) As explained in section 1.1, an "algorithm" ordinarily means a *method* for doing a task, but in data security it means the task itself; this is because the easiest way to define many tasks in data security is by specifying a method.

The MAA is a good example of this. As outlined in section 1.2, the MAA standard defines a long procedure, involving a prelude, a main loop, and a coda. This is an algorithm in the ordinary sense, but the important thing actually is not this procedure, but the MAC values that come out at the end of it. The real purpose of the standard is to define the *mapping*, ie which MAC should go with any given message and key. But the right MAC is deliberately made so obscure, that the only easy way to explain it is by giving a long recipe for calculating it.

It is conceivable that one could find a way of calculating the right MAC without following exactly the steps laid down in the MAA standard. This would be a different algorithm in the ordinary sense (a different *method*), but the same algorithm in the data security sense (the same *task*). In my view this would be an acceptable implementation.

The wording used in the MAA standard could be taken to mean that one must use the actual procedure it gives. For example, it says that "The main loop is a calculation which *shall be repeated* for each message block" (my italics). But if there is another way to calculate the MAC - without going through the main loop repeatedly - I think this should be allowed. More strikingly, the standard defines a function called MUL2, and states that this "shall not be used in the main loop"; this is in spite of the fact that the main loop uses a function called MUL2A, under conditions in which MUL2A always yields identical results to MUL2. MUL2A is usually a more efficient function in practice, taking advantage of the special conditions under which it is used; but MUL2 would still work.

I believe any such requirements concerning the *method* to be used are out of place. They cannot readily be expressed in a formal language either. I have therefore disregarded them: my LOTOS description defines what the right MAC value is in each case, not how it should be calculated. However, the LOTOS definitions largely *reflect* the procedure laid down in the MAA standard, and *hint* at this method of calculation. The VDM and Z descriptions [2, 3] take the same approach.

(b) The MAA standard states:

> Messages to be authenticated may originate as a bit string of any length. They shall be input to the algorithm as a sequence of 32 bit numbers, M_1, M_2 - M_n, of which there are n, called message blocks. The detail of how to pad out the last block M_n to 32 bits is not part of the algorithm but shall be defined in any application.

This indicates that the starting-point of the *algorithm* is a sequence of (32-bit) blocks. I argue that whether the message is previously a bit string, whether padding is needed, whether it is confined to the last block, and whether the last block is padded to the left or right (or otherwise), are matters left to the application, just as much as *how* to pad out the last block. Accordingly, my LOTOS description defines how to get from a block sequence to a MAC, but not whether or how to get from a bit string to a block sequence; I use the word "message" for a block sequence rather than a bit string. The question of padding can be mentioned in an informal commentary, rather than in the formal description.

In this, my approach differs from that of the VDM and Z descriptions [2, 3]. These include an explicit padding function to transform a bit string into a block sequence; the contents of the padding field are not specified (though padding is interpreted as being to the right of the last block). While I view this as out of place, it highlights an important point: VDM and Z *can* specify such a function, where the result is partly unspecified, whereas LOTOS cannot, as discussed in section 5.2.1.

The consequence of these two decisions is that my LOTOS description simply defines a *function*, in the strict mathematical sense. That is, it is equivalent to a look-up table, listing every possible combination of a message and a key, and giving a specific MAC for each such combination.

3.2 The choice of style

The following principles have guided my style:

(a) I have tried to support the simplest intuitive idea of the functions involved, in the clearest possible way. This contrasts with the VDM description [2], which sets out to follow the naming, structure etc of the MAA standard as closely as possible. I believe that certain functions can be presented more plainly by deviating from the presentational approach adopted in the standard, and I have not hesitated to do this.

One of the marks of a "simple" intuitive idea, in my view, is that the likely methods of implementation are easy to see.

(b) I have named sorts, operations etc in a style consistent with that of the standard library. For example, operations which perform elementary conversions to some sort are often given the same name as that sort (as with `Octet` in section 2.4 (e)).

(c) I have followed the naming, structure etc of the MAA standard where this did not conflict with (a) and (b).

3.3 The nature and use of the description

My description of the MAA is not a LOTOS specification, in the technical sense. The LOTOS standard names a large number of syntactic constructs, of which "specification" denotes a complete LOTOS text, while all the others form components of specifications. The MAA description is one such component, called "data-type-definitions". It consists of a set of data types, including a library declaration.

The central element of a true specification is a *behaviour specification*, defining the behaviour of a system. The "main" behaviour specification may invoke a hierarchy of *processes*, which contain their own behaviour specifications, defining sub-units of the overall behaviour. At all levels of the hierarchy it is possible to introduce data types; these provide values, and operations to manipulate them, which behaviour specifications can use.

The LOTOS standard defines the semantics of a specification in two phases:

(a) For any specification, the *static* semantics define a *canonical LOTOS specification.* As part of this process, every data type is translated into a *data-presentation*, which contains the sort names, operation names, functionalities and equations in a more abstract form; and the individual data-presentations corresponding to all the data types in the specification, from all levels, excluding parameterized types, are merged into a single data-presentation. This single data-presentation forms part of the canonical LOTOS specification.

(b) For any canonical LOTOS specification, the *dynamic* semantics define a choice of *structured labelled transition systems.* As part of this process, the (merged) data-presentation is translated into a many-sorted algebra, in the manner outlined in section 2.1.4. This many-sorted algebra embodies the entire collection of data values available to the system whose behaviour is defined; it forms part of each structured labelled transition system.

As explained in section 3.1, my MAA description defines a mathematical function. This has nothing to do with "behaviour" in the LOTOS sense, ie events in time. Instead, it corresponds to a LOTOS *operation.* This is why the description uses data types, but no behaviour specifications.

Although the MAA description is not a specification, it can be formally interpreted using *part* of the LOTOS semantics:

(a) Part of the static semantics will translate a set of data types into a set of data-presentations. This translation depends on the context of the data types, but it can be done for an "empty" context. In the LOTOS standard, the standard library is translated into a set of data-presentations in exactly this way; the MAA description can be treated similarly.

(b) Part of the dynamic semantics translates a data-presentation into a many-sorted algebra. While the main purpose is to apply this to the single data-presentation that combines all types in the specification, it can be applied to the data-presentation for any individual unparameterized type.

The important data type in the MAA description is the last one, called `AppliedMAA`. The preceding types define various concepts involved in the MAA, and are all imported into `AppliedMAA`. The dynamic semantics can be used to define one many-sorted algebra for each data-presentation, but the many-sorted algebra for `AppliedMAA` incorporates all the others and is the vital one.

This final many-sorted algebra contains a sort called `Block`, whose values represent all possible (32-bit) blocks. It also contains a sort called `AcceptableMessage`, whose values represent all acceptable messages; that is, there is one `AcceptableMessage` value for every possible string of 1 to 1 000 000 blocks. There is also a sort called `Pair`, whose values represent all possible pairs of blocks; these are the possible keys. The mathematical function referred to in section 3.1 is the operation:

```
Authenticator: Pair, AcceptableMessage -> Block
```

For any `Pair` value `Key` and any `AcceptableMessage` value `X`, `Authenticator(Key, X)` (a `Block` value) represents the correct MAC for that key and message.

One might have occasion to specify an actual system which performs MAA calculations. In natural language, this could be done by referring to the MAA standard, and adding implementation-specific details, eg concerning input and output procedures. An alternative would be to write a complete LOTOS specification; my MAA description could be a component of this text. Any extra data types which build on mine should follow the principles in the last part of section 2.2.2; the extra material will find its way into the single data-presentation in the canonical LOTOS specification, and influence the final many-sorted algebra; inappropriate operations or equations could change the MAA sorts and operations, so that they no longer reflect the MAA standard.

My LOTOS description is not designed to accompany the existing MAA standard; it is a fresh start, setting out to specify the MAA as plainly as LOTOS will allow, as indicated in section 3.2 (a). If used in earnest, as an alternative definitive text, it would have to be supported by an informal explanation. This explanation would describe the intuitive meaning of the sorts and operations, elucidate the implications of individual equations, and perhaps indicate methods of implementation. It might contain much material similar to that in the present MAA standard, though presented in a way better suited to the LOTOS approach.

I have not attempted to give the contents and format of an informal commentary. One would have to decide how much knowledge should be expected of a reader, and how much help is needed with using the LOTOS text. LOTOS allows comments to be inserted, but it may be better to provide a separate explanatory text instead; this approach would preserve a clarity of structure in the LOTOS text, allow greater freedom of presentation in the commentary, and allow different commentaries to be written for different kinds of reader. I have presented the LOTOS description without a commentary in the appendix, while the explanations in chapter 4 should serve readers of this report. Chapter 4 contains many points appropriate to a commentary.

Developing an MAA implementation from the LOTOS definition, as I envisage it, is the same as developing an implementation from the natural-language standard: one studies the text to find out what the implementation should do, and works out a way of doing it.

3.4 Testing the description

I have tried to use a LOTOS tool, developed by the ESPRIT project SEDOS [5], to check the description.

The data types first need to be placed in a proper specification. For this, I have specified a system that does nothing, though the MAA algebra (the sorts, values and operations) is established. The full text is formed by adding the line

```
specification MAA: noexit
```

at the top, and the following at the bottom:

```
behaviour
    stop

endspec
```

The tool reported no syntax errors, but a long list of static semantic errors. However, I have established that there are faults in the tool, and these appear to account for all the error reports. This has prevented evaluation of sample expressions to test the operations.

However, I expected insurmountable problems in evaluating expressions in any case. These result from the awkward representation of large numbers, and the difficulty of determining when ground terms are interderivable. I would expect these problems to arise with *any* LOTOS tool.

It might be considered important to have a description which *can* be thoroughly checked using a tool; this could be done for the MAA with considerable rewriting of my description. A new definition of natural numbers would have to be written, generating them more efficiently than by repeatedly applying `Succ` to `0`; this would have to be used in place of the standard library type `NaturalNumber`. The standard library types `NonEmptyString`, `Bit`, `Octet`, `BitNatRepr` and `DecNatRepr`, which import `NaturalNumber`, would also have to be "hand rewritten", so that they use the new natural numbers. Interderivability of ground terms can be algorithmically determined if the style of equations is restricted. I have made extensive use of conditional equations in which some of the conditions define *implicit* calculations; these would have to be rewritten as explicit calculations. This would involve introducing a lot of extra operations, many of which would represent partial functions. (This last point is touched on in section 5.1.1, which explains that partial functions present problems.) The resulting description would be much longer and intuitively less plain.

4 Describing the MAA - the specification details

The MAA description is presented in a very piecemeal fashion in this chapter, with explanations. To see how the various pieces of LOTOS are put together, refer to the appendix.

4.1 Blocks and the basic functions manipulating them

Fundamental to the MAA are "32-bit unsigned integers", called *blocks* in this report. Blocks undergo a mixture of logical and arithmetic manipulations, to achieve effective "scrambling" of information. The basic ideas associated with blocks are described in the first data type for MAA, called `BlockFunctions`.

The functions defined in the MAA standard and covered by type `BlockFunctions` are denoted by CYC, AND, OR, XOR, ADD, CAR, MUL1, MUL2 and MUL2A. I have not defined them all explicitly in LOTOS.

4.1.1 The concept of a block

There are two ways of viewing a block:

(a) It can be viewed as a string of 32 bits. The MAA acts, fundamentally, on information represented in a binary form; a block can be thought of as a basic unit of such information. This view of a block is therefore the primary one. It is the relevant view when blocks are subjected to logical and other operations which manipulate individual bits.

(b) It can be viewed as a number, ranging from 0 to 2^{32}-1. This is relevant when blocks are subjected to arithmetic operations (addition and multiplication). This is therefore an important secondary view of a block.

These views are captured in the standard library type `BitNatRepr`: this provides the sort `BitString`, containing bit strings, and the operation `NatNum`, which identifies, for any given bit string, the corresponding natural number. Type `BitNatRepr` is therefore imported into type `BlockFunctions`; this provides a valuable initial resource.

However, the sort `BitString` includes bit strings of all sizes. It is essential to have a sort containing 32-bit strings only, because many MAA operations are restricted to these. As explained in section 2.1.1, an operation in LOTOS is a total function; this means that if one of the arguments is of sort `BitString`, then *any* bit string can be used. So I have defined a sort `Block`, which consists of all possible blocks, generated by a constructor operation:

```
Block:
            Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit,
            Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit,
            Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit,
            Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit
         ->
            Block
```

This is analogous to the standard library sort `Octet`, which consists, in effect, of all 8-bit strings.

Some of the equations use 32 variables for the individual bits in a block. For compactness, two sets of such variables are defined "once and for all" at the outset:

```
forall
     x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
     x9,  x10, x11, x12, x13, x14, x15, x16,
     x17, x18, x19, x20, x21, x22, x23, x24,
     x25, x26, x27, x28, x29, x30, x31, x32,
     y1,  y2,  y3,  y4,  y5,  y6,  y7,  y8,
     y9,  y10, y11, y12, y13, y14, y15, y16,
     y17, y18, y19, y20, y21, y22, y23, y24,
     y25, y26, y27, y28, y29, y30, y31, y32: Bit
```

This contrasts with all other variables in the MAA description, which are local to a small group of equations, being defined *after* the associated "`ofsort`" statement; the latter style is clearest, because the total collection of both equations and variables, in each data type, is fairly large and varied.

To take advantage of the operations provided for bit strings, applying them to blocks, it is necessary to recognize blocks as bit strings. This is done using an operation

```
BitString: Block -> BitString
```

which "converts" a block to the equivalent `BitString` value:

```
BitString
  (
    Block
      (
        x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
        x9,  x10, x11, x12, x13, x14, x15, x16,
        x17, x18, x19, x20, x21, x22, x23, x24,
        x25, x26, x27, x28, x29, x30, x31, x32
      )
  )
  =
    Bit(x1) + x2 + x3 + x4 + x5 + x6  + x7  + x8  +
    x9  + x10 + x11 + x12 + x13 + x14 + x15 + x16 +
    x17 + x18 + x19 + x20 + x21 + x22 + x23 + x24 +
    x25 + x26 + x27 + x28 + x29 + x30 + x31 + x32;
```

Here, + is the operation which appends a bit to a bit string. In the absence of brackets, the operations are evaluated from left to right.

This definition has an important consequence: the first argument to the `Block` operation is regarded as the first bit of the string, which means it is the most significant bit when the block is viewed as a number. While this is the natural interpretation of the arguments to `Block`, this equation makes it explicit.

Using operations

```
_eq_, _ne_: Block, Block -> Bool
```

`X eq Y` and `X ne Y` give the status of the relations "`X` is equal to `Y`" and "`X` is not equal to `Y`". These operations are already defined for bit strings, and are easy to transfer to blocks:

```
forall X, Y: Block

X eq Y = BitString(X) eq BitString(Y);

X ne Y = BitString(X) ne BitString(Y);
```

4.1.2 Logical operations

CYC(X) denotes the result of a one-bit cyclic left shift of X. This is represented by an operation

```
CYC: Block -> Block
```

where:

```
CYC
  (
    Block
      (
        x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
        x9,  x10, x11, x12, x13, x14, x15, x16,
        x17, x18, x19, x20, x21, x22, x23, x24,
        x25, x26, x27, x28, x29, x30, x31, x32
      )
  )
  =
    Block
      (
        x2,  x3,  x4,  x5,  x6,  x7,  x8,  x9,
        x10, x11, x12, x13, x14, x15, x16, x17,
        x18, x19, x20, x21, x22, x23, x24, x25,
        x26, x27, x28, x29, x30, x31, x32, x1
      );
```

While a much more compact definition is possible, this one clearly shows the movement of bits which is the essence of this function.

XOR(X, Y) denotes the result of a bitwise "exclusive OR" operation. This can be defined with the help of a separate "exclusive OR" operation for individual bits

```
_xor_: Bit, Bit -> Bit
```

where:

```
0 xor 0 = 0;
0 xor 1 = 1;
1 xor 0 = 1;
1 xor 1 = 0;
```

The XOR function for blocks is represented by an operation

```
XOR: Block, Block -> Block
```

where:

```
XOR
  (
    Block
      (
        x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
        x9,  x10, x11, x12, x13, x14, x15, x16,
        x17, x18, x19, x20, x21, x22, x23, x24,
        x25, x26, x27, x28, x29, x30, x31, x32
      ),
    Block
      (
        y1,  y2,  y3,  y4,  y5,  y6,  y7,  y8,
        y9,  y10, y11, y12, y13, y14, y15, y16,
        y17, y18, y19, y20, y21, y22, y23, y24,
        y25, y26, y27, y28, y29, y30, y31, y32
      )
  )
  =
    Block
      (
        x1  xor y1,  x2  xor y2,  x3  xor y3,  x4  xor y4,
        x5  xor y5,  x6  xor y6,  x7  xor y7,  x8  xor y8,
        x9  xor y9,  x10 xor y10, x11 xor y11, x12 xor y12,
        x13 xor y13, x14 xor y14, x15 xor y15, x16 xor y16,
        x17 xor y17, x18 xor y18, x19 xor y19, x20 xor y20,
        x21 xor y21, x22 xor y22, x23 xor y23, x24 xor y24,
        x25 xor y25, x26 xor y26, x27 xor y27, x28 xor y28,
        x29 xor y29, x30 xor y30, x31 xor y31, x32 xor y32
      );
```

Again, while long-winded, this definition clearly expresses the idea of a bitwise operation.

AND(X, Y) and OR(X, Y), similarly, denote the results of bitwise AND and OR operations. They could be defined in the same way as XOR. However, they have a very limited use in the MAA, appearing only in the following lines (part of the main loop):

```
F := OR(F, A);          G := OR(G, B);
F := AND(F, C);         G := AND(G, D);
```

A, B, C and D are constants, defined in hexadecimal as 0204 0801, 0080 4021, BFEF 7FDF and 7DFE FBFF, respectively. The effect of OR with a constant is to change the bits in certain fixed positions to 1, while AND with a constant changes certain bits to 0. I have therefore chosen to define two operations

```
FIX1, FIX2: Block -> Block
```

which encapsulate the specific transformations applied to F and G, respectively, by the above two lines. This form of definition shows *what* is done at this stage of the MAA, as opposed to *how* it is done. It is also much shorter than defining AND, OR and the four constants:

```
FIX1
  (
    Block
      (
        x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
        x9,  x10, x11, x12, x13, x14, x15, x16,
        x17, x18, x19, x20, x21, x22, x23, x24,
        x25, x26, x27, x28, x29, x30, x31, x32
      )
  )
  =
    Block
      (
        x1,  0,   x3,  x4,  x5,  x6,  1,   x8,
        x9,  x10, x11, 0,   x13, 1,   x15, x16,
        0,   x18, x19, x20, 1,   x22, x23, x24,
        x25, x26, 0,   x28, x29, x30, x31, 1
      );

FIX2
  (
    Block
      (
        x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
        x9,  x10, x11, x12, x13, x14, x15, x16,
        x17, x18, x19, x20, x21, x22, x23, x24,
        x25, x26, x27, x28, x29, x30, x31, x32
      )
  )
  =
    Block
      (
        0,   x2,  x3,  x4,  x5,  x6,  0,   x8,
        1,   x10, x11, x12, x13, x14, x15, 0,
        x17, 1,   x19, x20, x21, 0,   x23, x24,
        x25, x26, 1,   x28, x29, x30, x31, 1
      );
```

4.1.3 Arithmetic operations

The MAA standard defines five functions related to addition and multiplication. Since a block can be viewed as a 32-bit number, two blocks can be added to give a 33-bit number, or multiplied to give a 64-bit number. (33 or 64 bits, here, are what is enough to represent *any* possible sum or product.)

The MAA functions use the 33-bit or 64-bit result to define further blocks. ADD(X, Y) denotes the result of adding X and Y and discarding the final carry; that is, it denotes the least significant 32 bits of the sum. Meanwhile, CAR(X, Y) denotes the final carry from adding X and Y; that is, it has the value of the most significant bit of the 33-bit sum. So if ADD(X, Y) is denoted by `S` (a

`Block` value) while CAR(X, Y) is denoted by `C` (a `Bit` value), then the sum of X and Y is represented by the 33-bit string `C + BitString(S)`. The three multiplication-based functions all use the product of X and Y, with the upper (most significant) half denoted by U and the lower half by L. If these are represented by `Block` values `U` and `L`, then the product of X and Y is represented by the 64-bit string `BitString(U) ++ BitString(L)`.

The actual numbers represented by the 32-bit string formed of `X` (a `Block` value), the 33-bit string formed of `C` and `S`, and the 64-bit string formed of `U` and `L`, are denoted by:

```
NatNum(BitString(X))
NatNum(C + BitString(S))
NatNum(BitString(U) ++ BitString(L))
```

These three expressions would be much simpler if `Block` values did not have to be explicitly converted to `BitString` values. To make this possible, I have defined three operations which treat a `Block` value as though it were a `BitString` value:

```
NatNum: Block -> Nat
_+_:    Bit, Block -> BitString
_++_:   Block, Block -> BitString
```

These operations are easy to transfer from bit strings to blocks:

```
forall X: Block

NatNum(X) = NatNum(BitString(X));
```

and

```
forall X, Y: Block, b: Bit

b + X = b + BitString(X);

X ++ Y = BitString(X) ++ BitString(Y);
```

Now the above three numbers can be denoted by:

```
NatNum(X)
NatNum(C + S)
NatNum(U ++ L)
```

(The first expression uses the new `NatNum`, while the others use the old `NatNum` with the new + and ++.)

The function MUL1 begins by multiplying X and Y to give U and L. The rest of the calculation is defined in the MAA standard by the following assignments:

```
S := ADD(U, L);
C := CAR(U, L);
MUL1(X, Y) := ADD(S, C).
```

This is represented by an operation:

```
MUL1: Block, Block -> Block
```

The key technique I have used to define MUL1, and various other multi-step calculations, is the conditional equation; the conditions are those established by the calculation steps. I use variables to represent both "input" values, and values generated during the calculation. To meet the conditions, the "input" variables can have any values of the right sorts, but the other variables are forced to have the particular values that result from the actual calculation. In the case of MUL1, the "input" values are `X` and `Y`, while generated values are `U`, `L`, `S`, `C` and the final result, which I denote by `P`:

```
forall X, Y, U, L, S, P: Block, C: Bit

    NatNum(X) * NatNum(Y) = NatNum(U ++ L),
    NatNum(U) + NatNum(L) = NatNum(C + S),
    NatNum(S) + NatNum(C) = NatNum(P)
  =>
MUL1(X, Y) = P;
```

Given `X` and `Y`, the first condition establishes what the numeric value of `U ++ L` has to be; but this implicitly fixes `U` and `L`, because they are (32-bit) blocks, and there is only one 64-bit string that can represent the specified numeric value. In a similar way, the second condition fixes `S` and `C`, and the third fixes `P`. The third condition works because the final carry from adding S and C is 0,[2] and so the least significant 32 bits are, in fact, the whole sum. The important outcome is that, for any `X` and `Y`, the conditions fix `P`, and so `MUL1(X, Y)` is defined to be that `P` value and no other.

The function MUL2 begins, again, by multiplying X and Y to give U and L. The rest of the calculation, this time, is:

```
D := ADD(U, U);
E := CAR(U, U);
F := ADD(D, 2E);
S := ADD(F, L);
C := CAR(F, L);
MUL2(X, Y) := ADD(S, 2C).
```

Here, E and C are multiplied by 2; U, in effect, is also multiplied by 2 to give D and E. 2 is denoted in type `BitNatRepr` by `Succ(NatNum(1))` (where `1` is of sort `Bit`), but it is convenient to define a constant

```
2: -> Nat
```

where:

```
2 = Succ(NatNum(1));
```

2 The maximum sum of two blocks consists of 32 ones followed by a zero. C and S form the sum of U and L; so if C is non-zero, S cannot consist of 32 ones, and C can be added without overflowing 32 bits.

MUL2 is represented by an operation

```
MUL2: Block, Block -> Block
```

where:

```
forall X, Y, U, L, S, P, D: Block, C, E: Bit

    NatNum(X) * NatNum(Y)                        = NatNum(U ++ L),
    2 * NatNum(U)                                = NatNum(E + D),
    NatNum(D) + (2 * NatNum(E)) + NatNum(L) = NatNum(C + S),
    NatNum(S) + (2 * NatNum(C))                  = NatNum(P)
  =>
MUL2(X, Y) = P;
```

F does not appear as a variable; its numeric value is given by the expression

```
NatNum(D) + (2 * NatNum(E))
```

which is used in the third condition. This works because the final carry from adding D and 2E is 0,[3] and so the least significant 32 bits are the whole sum. Likewise, the fourth condition works because the final carry from adding S and 2C is 0.[4]

The function MUL2A is a simpler way of calculating MUL2. This works when either X or Y begins with a zero bit, but can give a different result otherwise. Under the condition just mentioned, the calculation can be simplified because the product of X and Y begins with a zero bit, and so E is always 0. As before, X and Y are multiplied to give U and L; the rest of the calculation is:

```
D := ADD(U, U);
S := ADD(D, L);
C := CAR(D, L);
MUL2A(X, Y) := ADD(S, 2C).
```

These steps also define a value for MUL2A when X and Y do not satisfy the above condition; only, this value may differ from that of MUL2. In such cases, E is not necessarily 0 but is unused. MUL2A, as it acts on arbitrary blocks X and Y, is represented by an operation

```
MUL2A: Block, Block -> Block
```

[3] U cannot consist of 32 ones, because then the product of X and Y would exceed the maximum product of two blocks. So 2U cannot consist of 32 ones followed by a zero; so if E is non-zero, D cannot consist of 31 ones followed by a zero, and 2E can be added without overflowing 32 bits.

[4] D and 2E are both even, and so their sum F is also even, and less than the maximum block. So the sum of F and L cannot amount to 32 ones followed by a zero; so if C is non-zero, S cannot consist of 31 ones followed by a zero, and 2C can be added without overflowing 32 bits.

where:

```
forall X, Y, U, L, S, P, D: Block, C, E: Bit

    NatNum(X) * NatNum(Y)         = NatNum(U ++ L),
    2 * NatNum(U)                 = NatNum(E + D),
    NatNum(D) + NatNum(L)         = NatNum(C + S),
    NatNum(S) + (2 * NatNum(C))   = NatNum(P)
  =>
MUL2A(X, Y) = P;
```

As before, the fourth condition works because the final carry from adding S and 2C is 0.[5]

The function CAR has no use in the MAA standard except in defining MUL1, MUL2 and MUL2A. Since the LOTOS definitions of MUL1, MUL2 and MUL2A do not use CAR explicitly, I have not defined it.

The function ADD, however, has a separate use in the main loop, and is represented by an operation

```
ADD: Block, Block -> Block
```

where:

```
forall X, Y, S: Block, C: Bit

    NatNum(X) + NatNum(Y) = NatNum(C + S)
  =>
ADD(X, Y) = S;
```

4.2 Pairs of blocks and the conditioning functions

The key used in the MAA consists of a pair of blocks, and blocks fall naturally into pairs elsewhere in the algorithm. The prelude uses a function called BYT to "condition" the key and other pairs of blocks; this conditioning involves certain adjustments to prevent long strings of zeros or ones. BYT manipulates individual bytes within a block, ie octets. A related function PAT generates a further octet. These ideas are described in the second data type for MAA, called `ConditioningFunctions`.

4.2.1 Concepts related to block pairs and octets

The ideas of a block and an octet are essential to BYT and PAT. Type `ConditioningFunctions` therefore imports type `BlockFunctions`, and the standard library type `Octet`. Type `BlockFunctions` has already established the relationship between blocks and the equivalent `BitString` values; type `ConditioningFunctions` establishes a

[5] D is even, and therefore less than the maximum block. So the sum of D and L cannot amount to 32 ones followed by a zero; so if C is non-zero, S cannot consist of 31 ones followed by a zero, and 2C can be added without overflowing 32 bits.

similar relationship between octets and BitString values. This involves using the material from type BitNatRepr again; although this is imported via BlockFunctions, I have imported it again explicitly into ConditioningFunctions, because it has fresh uses.

The sort Pair consists of all pairs of blocks, generated by a constructor operation:

```
Pair: Block, Block -> Pair
```

Any Pair value can be expressed as Pair(X, Y) for only one X and Y; thus it amounts to a "record" of the X and Y that have been "placed" in it. The two blocks in a pair are meant to be viewed as separate variables, not as juxtaposed to form a 64-bit string. But they are not "mixed up", because Pair(X, Y) and Pair(Y, X) are distinct (assuming X and Y are different).

Like a block, an octet can be viewed in two ways: a string of 8 bits, or a number from 0 to 255. As before, these views are captured in type BitNatRepr, but octets need to be recognized as bit strings; this is done using an operation

```
BitString: Octet -> BitString
```

which "converts" an octet to the equivalent BitString value:

```
forall b1, b2, b3, b4, b5, b6, b7, b8: Bit

BitString(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) =
    Bit(b1) + b2 + b3 + b4 + b5 + b6 + b7 + b8;
```

This definition implies that the first argument to the Octet operation is regarded as the first bit of the string, and therefore the most significant bit of a number. There is actually nothing in the standard library type Octet to indicate this interpretation; but it is the natural interpretation, and I have hereby explicitly adopted it for MAA.

As with blocks in section 4.1.3, certain expressions can be simplified if Octet values do not have to be explicitly converted to BitString values. In this case, I have defined one operation which treats an Octet value as though it were a BitString value:

```
NatNum: Octet -> Nat
```

Again, this is easy to transfer from bit strings to octets:

```
forall B: Octet

NatNum(B) = NatNum(BitString(B));
```

I have defined a bitwise "exclusive OR" operation between octets:

```
_xor_: Octet, Octet -> Octet
```

This uses the "exclusive OR" defined in type `BlockFunctions` for individual bits:

```
forall
    x1, x2, x3, x4, x5, x6, x7, x8,
    y1, y2, y3, y4, y5, y6, y7, y8: Bit

Octet(x1, x2, x3, x4, x5, x6, x7, x8) xor
Octet(y1, y2, y3, y4, y5, y6, y7, y8)
  =
    Octet
      (
        x1 xor y1, x2 xor y2, x3 xor y3, x4 xor y4,
        x5 xor y5, x6 xor y6, x7 xor y7, x8 xor y8
      );
```

4.2.2 The functions BYT and PAT

The MAA standard uses a square-bracket notation for concatenation. For example, [X, Y, Z] would denote the result of concatenating X, Y and Z in that order; X, Y and Z are implicitly regarded as bit strings (in which the first bit is "most significant").

The functions BYT and PAT act on a pair of blocks X and Y, which, for convenience, are concatenated into the 64-bit string [X, Y]. The function results are denoted by BYT[X, Y] and PAT[X, Y]. BYT[X, Y] is another pair of blocks (the "conditioned" values of X and Y), again concatenated into a 64-bit string, while PAT[X, Y], in essence, is an octet. These two functions are closely related, and are defined in the MAA standard as two results of a single calculation: the value of PAT is an encapsulation of the adjustments made by BYT.

The MAA standard presents a Pascal statement to define the calculation. The individual bytes making up X and Y are denoted by B_0 etc, as defined by the formula:

$$[X, Y] = [B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7]$$

Each byte B_i is stored in an integer array element B[i]. "Conditioned" bytes B'_0 etc are likewise stored in an array with non-standard identifier B'. Array B contains the bytes B_i before the calculation begins, while array B' is filled with the bytes B'_i during the calculation. The calculation is:

```
    begin
    P := 0;
    for i := 0 to 7 do
    begin
        P := 2 * P;
        if B[i] = 0 then
        begin
            P := P + 1;
            B'[i] := P
        end
    else if B[i] = 255 then
        begin
            P := P + 1;
            B'[i] := 255 - P
        end
    else
    B'[i] := B[i];
    end
end;
```

The relevant results are:

```
BYT[X, Y] = [B'0, B'1, B'2, B'3, B'4, B'5, B'6, B'7]
PAT[X, Y] = P
```

The rest of this section unravels this Pascal statement to establish the underlying order, and gives the LOTOS description of that order.

The essence of the statement is a loop which executes eight times, with i taking the values 0 to 7, thereby indexing successive elements of B and B'. In each execution, the only variables used are B[i], B'[i] and P. B[i] never changes, B'[i] is assigned "once and for all", and P is modified in every execution of the loop. The only action outside the loop is to initialize P.

In each execution of the loop, the action is controlled by the value of B[i]. When B[i] is neither 0 nor 255, the value of B'[i] becomes the same as B[i]. But when B[i] is 0 or 255, B'[i] becomes something different; I refer to this as an "adjustment". The significance of 255 is that it is the octet consisting of 8 ones. Using an operation

```
NeedAdjust: Octet -> Bool
```

`NeedAdjust(B)` gives the status of the condition "`B` needs an adjustment", where `B` represents a B[i] value:

```
forall B: Octet

NeedAdjust(B) =
     (B eq Octet(0, 0, 0, 0, 0, 0, 0, 0)) or
     (B eq Octet(1, 1, 1, 1, 1, 1, 1, 1));
```

The changes made to P in each loop execution depend on whether B[i] needs an adjustment. When there is no adjustment, P is just multiplied by 2; this appends a zero to its binary representation. When there is an adjustment, P is multiplied by 2 and then increased by 1; these together append a one to its binary representation. The bit appended to P, determined by the value of B[i], is given by an operation

```
AdjustCode: Octet -> Bit
```

where:

```
forall B: Octet

NeedAdjust(B) =>      AdjustCode(B) = 1;
not(NeedAdjust(B)) => AdjustCode(B) = 0;
```

Note the form of these conditional equations: a condition may be an expression of sort `Bool` (or any other sort containing a constant `true`) rather than another equation. The condition is met when the expression is equal to `true`.

The eight loop executions generate eight bits at the end of P, corresponding to the eight B[i] values. These eight bits form the octet PAT[X, Y]. (Any preceding bits of P are zero, because of initialization, and can be omitted.) Rather than the concatenation [X, Y] (one 64-bit string), I have used the more abstract `Pair(X, Y)` (two 32-bit blocks). Then PAT is represented by an operation:

```
PAT: Pair -> Octet
```

The bytes B_0 to B_7 are represented by `Octet` variables `B1` to `B8` (following the numbering style of the standard library):

```
forall X, Y: Block, B1, B2, B3, B4, B5, B6, B7, B8: Octet

    BitString(X) =
        BitString(B1) ++ BitString(B2) ++
        BitString(B3) ++ BitString(B4),
    BitString(Y) =
        BitString(B5) ++ BitString(B6) ++
        BitString(B7) ++ BitString(B8)
  =>
PAT(Pair(X, Y)) =
    Octet
      (
        AdjustCode(B1), AdjustCode(B2),
        AdjustCode(B3), AdjustCode(B4),
        AdjustCode(B5), AdjustCode(B6),
        AdjustCode(B7), AdjustCode(B8)
      );
```

Given `X` and `Y`, the conditions fix the eight octet values, and therefore the value of `PAT(Pair(X, Y))`.

When B[i] needs an adjustment, this consists of replacing 0 by P, or 255 by (255 - P). P can be regarded as an octet, whose value is studied shortly. Since 255 is the octet consisting of 8 ones, (255 - P) gives the octet with a one in each position where P has a zero, and a zero where P has a one. There are two simple ways of viewing the adjustment:

(a) B'[i] becomes either the octet P or its logical inversion. The (eight identical) bits of B[i] form a control flag, where "ones" imply "invert".

(b) B'[i] becomes a copy of B[i] in which certain bits are inverted. The ones in P mark the positions in which to invert.

Both views are captured in the symmetrical statement that (when there is an adjustment) B'[i] becomes the bitwise "exclusive OR" result from B[i] and P. Using an operation

```
Adjust: Octet, Octet -> Octet
```

if B[i] and P are represented by `B` and `P`, then `Adjust(B, P)` gives the value for B'[i] (whether or not there is an adjustment):

```
forall B, P: Octet

NeedAdjust(B) =>      Adjust(B, P) = B xor P;
not(NeedAdjust(B)) => Adjust(B, P) = B;
```

The value P used to adjust B[i] has the arbitrary character common in data security: it is an intermediate value in the calculation of PAT[X, Y], and is different for each of the eight B[i] values. As already seen, each loop execution appends one bit to its binary representation; this, in effect, shifts the existing bits to the left. This modification precedes the adjustment of B[i].

The eight B'[i] values make up BYT[X, Y], which may be denoted by [Xc, Yc] (the "conditioned" values of X and Y). As before, I have used `Pair(X, Y)` rather than [X, Y] as the argument; likewise, I have used `Pair(Xc, Yc)` rather than [Xc, Yc] for the result. Then BYT is represented by an operation:

```
BYT: Pair -> Pair
```

As before, the bytes B_0 to B_7 are represented by `Octet` variables `B1` to `B8`; likewise, the "conditioned" bytes B'_0 to B'_7 are represented by `Octet` variables `Bc1` to `Bc8`:

```
forall
    X, Y, Xc, Yc: Block,
    B1,  B2,  B3,  B4,  B5,  B6,  B7,  B8,
    Bc1, Bc2, Bc3, Bc4, Bc5, Bc6, Bc7, Bc8: Octet,
    p1,  p2,  p3,  p4,  p5,  p6,  p7,  p8: Bit

    BitString(X) =
        BitString(B1) ++ BitString(B2) ++
        BitString(B3) ++ BitString(B4),
    BitString(Y) =
        BitString(B5) ++ BitString(B6) ++
        BitString(B7) ++ BitString(B8),
    PAT(Pair(X, Y)) = Octet(p1, p2, p3, p4, p5, p6, p7, p8),
    Adjust(B1, Octet(0,  0,  0,  0,  0,  0,  0,  p1)) = Bc1,
    Adjust(B2, Octet(0,  0,  0,  0,  0,  0,  p1, p2)) = Bc2,
    Adjust(B3, Octet(0,  0,  0,  0,  0,  p1, p2, p3)) = Bc3,
    Adjust(B4, Octet(0,  0,  0,  0,  p1, p2, p3, p4)) = Bc4,
    Adjust(B5, Octet(0,  0,  0,  p1, p2, p3, p4, p5)) = Bc5,
    Adjust(B6, Octet(0,  0,  p1, p2, p3, p4, p5, p6)) = Bc6,
    Adjust(B7, Octet(0,  p1, p2, p3, p4 ,p5, p6, p7)) = Bc7,
    Adjust(B8, Octet(p1, p2, p3, p4, p5, p6, p7, p8)) = Bc8,
    BitString(Xc) =
        BitString(Bc1) ++ BitString(Bc2) ++
        BitString(Bc3) ++ BitString(Bc4),
    BitString(Yc) =
        BitString(Bc5) ++ BitString(Bc6) ++
        BitString(Bc7) ++ BitString(Bc8)
  =>
BYT(Pair(X, Y)) = Pair(Xc, Yc);
```

Given `X` and `Y`, the first three conditions fix `B1` to `B8`, and (since `PAT` is already defined) `p1` to `p8`; the next eight conditions fix `Bc1` to `Bc8`; and the last two conditions fix `Xc` and `Yc`. In this way, the value of `BYT(Pair(X, Y))` is fixed.

4.3 "The algorithm" proper

Section 1.2 distinguishes a lower-level function, called "the algorithm", and a higher-level function, which is "the mode of operation" of that algorithm. I take "MAA" ("message authenticator algorithm") to mean, technically, the lower-level function, while "message authentication" is based on the higher-level function.

4.3.1 Messages

"The algorithm" acts on a sequence of one or more blocks, which form a *message*. Because of the mode of operation, this "message" is sometimes not the actual message to be authenticated, but a derived sequence of blocks; but "message" is still a convenient term for it. These messages, and actual messages, can both be viewed as strings, whose elements are blocks.

The third data type for MAA simply defines the concept of a message. Called `Message`, the type is an actualization of the standard library type `NonEmptyString`:

```
type Message is

    NonEmptyString actualizedby BlockFunctions using

        sortnames
            Bool for FBool
            Block for Element
            Message for NonEmptyString

        opnnames
            Message for String

endtype
```

The important parameter, `Element`, is thus defined to be the sort `Block` imported from `BlockFunctions`. That is, the elements of strings will be blocks. The associated operation parameters `eq` and `ne` are those defined in `BlockFunctions`. The technical parameters `FBool`, `true` and `not` are defined to be the `Bool`, `true` and `not` imported from `Boolean` via `BlockFunctions`.

The sort `NonEmptyString`, defined in *type* `NonEmptyString`, is renamed `Message`. So `Message` consists of all strings of one or more blocks, ie all messages. The operation `String`, which generates strings of one element, is also renamed `Message`. So the message consisting of just the block `X` is denoted by `Message(X)`. All other operations defined by type `NonEmptyString` keep the same names in type `Message`.

4.3.2 The MAA calculations

The calculations involved in the MAA proper ("the algorithm") are described in the fourth data type for MAA, called `BasicMAA`. Types `BlockFunctions` and `ConditioningFunctions` are imported, because their contents are central to the calculations. Type `Message` is also imported, since the MAA acts on a message.

Type `ConditioningFunctions` defines the sort `Pair`, consisting of pairs of blocks. Some of the calculations in the MAA yield two pairs, or three pairs, of blocks; but a LOTOS operation can yield only one result. It is therefore convenient to define a single object which contains two pairs, or three pairs, just as a pair is itself a single object containing two blocks. The sorts `TwoPairs` and `ThreePairs` consist of these objects, generated by constructor operations:

```
TwoPairs:   Pair, Pair -> TwoPairs
ThreePairs: Pair, Pair, Pair -> ThreePairs
```

The first part of the calculation is the *prelude*. The input to this is the key, consisting of the blocks J and K. If these are represented by `Block` values `J` and `K`, the key can be represented by `Pair(J, K)`. From this the prelude generates six blocks denoted by X_0, Y_0, V_0, W, S and T. These, in the way they are formed and later used, fall naturally into three pairs. If they are represented by `Block` values `X0`, `Y0`, `V0`, `W`, `S` and `T`, the result of the prelude can be represented by:

```
ThreePairs(Pair(X0, Y0), Pair(V0, W), Pair(S, T))
```

Then the prelude is represented by an operation:

```
Prelude: Pair -> ThreePairs
```

The MAA standard specifies the prelude as follows:

```
$[J_1, K_1]$ := BYT[J, K];
P := PAT[J, K];
Q := (1 + P) * (1 + P).

$J1_2$ := MUL1($J_1$, $J_1$);      $J2_2$ := MUL2($J_1$, $J_1$);
$J1_4$ := MUL1($J1_2$, $J1_2$);  $J2_4$ := MUL2($J2_2$, $J2_2$);
$J1_6$ := MUL1($J1_2$, $J1_4$);  $J2_6$ := MUL2($J2_2$, $J2_4$);
$J1_8$ := MUL1($J1_2$, $J1_6$);  $J2_8$ := MUL2($J2_2$, $J2_6$).
        $H_4$ := XOR($J1_4$, $J2_4$);
        $H_6$ := XOR($J1_6$, $J2_6$);
        $H_8$ := XOR($J1_8$, $J2_8$).

$K1_2$ := MUL1($K_1$, $K_1$);     $K2_2$ := MUL2($K_1$, $K_1$);
$K1_4$ := MUL1($K1_2$, $K1_2$); $K2_4$ := MUL2($K2_2$, $K2_2$);
$K1_5$ := MUL1($K_1$, $K1_4$);  $K2_5$ := MUL2($K_1$, $K2_4$);
$K1_7$ := MUL1($K1_2$, $K1_5$); $K2_7$ := MUL2($K2_2$, $K2_5$);
$K1_9$ := MUL1($K1_2$, $K1_7$); $K2_9$ := MUL2($K2_2$, $K2_7$).
        H' := XOR($K1_5$, $K2_5$);
        $H_5$ := MUL2(H', Q);
        $H_7$ := XOR($K1_7$, $K2_7$);
        $H_9$ := XOR($K1_9$, $K2_9$).

$[X_0, Y_0]$ := BYT[$H_4$, $H_5$];
$[V_0, W]$ := BYT[$H_6$, $H_7$];
[S, T] := BYT[$H_8$, $H_9$].
```

Given an "input" value `Key`, all other values can be fixed by a series of conditions:

```
forall
    Key: Pair,
    P: Octet,
    Jc, Kc, Q,
    J1_2, J1_4, J1_6, J1_8, J2_2, J2_4, J2_6, J2_8,
    K1_2, K1_4, K1_5, K1_7, K1_9, K2_2, K2_4, K2_5, K2_7, K2_9,
    H4, H5, H6, H7, H8, H9: Block

    BYT(Key) = Pair(Jc, Kc),
    PAT(Key) = P,
    (NatNum(1) + NatNum(P)) * (NatNum(1) + NatNum(P)) = NatNum(Q),
    MUL1(Jc, Jc)     = J1_2,
    MUL1(J1_2, J1_2) = J1_4,
    MUL1(J1_2, J1_4) = J1_6,
    MUL1(J1_2, J1_6) = J1_8,
    MUL2(Jc, Jc)     = J2_2,
    MUL2(J2_2, J2_2) = J2_4,
    MUL2(J2_2, J2_4) = J2_6,
    MUL2(J2_2, J2_6) = J2_8,
    XOR(J1_4, J2_4) = H4,
    XOR(J1_6, J2_6) = H6,
    XOR(J1_8, J2_8) = H8,
    MUL1(Kc, Kc)     = K1_2,
    MUL1(K1_2, K1_2) = K1_4,
    MUL1(Kc, K1_4)   = K1_5,
    MUL1(K1_2, K1_5) = K1_7,
    MUL1(K1_2, K1_7) = K1_9,
    MUL2(Kc, Kc)     = K2_2,
    MUL2(K2_2, K2_2) = K2_4,
    MUL2(Kc, K2_4)   = K2_5,
    MUL2(K2_2, K2_5) = K2_7,
    MUL2(K2_2, K2_7) = K2_9,
    MUL2(XOR(K1_5, K2_5), Q) = H5,
    XOR(K1_7, K2_7)          = H7,
    XOR(K1_9, K2_9)          = H9
  =>
Prelude(Key) =
    ThreePairs
      (
        BYT(Pair(H4, H5)),
        BYT(Pair(H6, H7)),
        BYT(Pair(H8, H9))
      );
```

The last three assignments in the informal specification are reflected in the three argument values in the final expression, rather than in the conditions. (Remember which way the six blocks X_0 etc are embodied within a `ThreePairs` value.) Note that the constant `1`, appearing in the third condition, is of sort `Bit`, imported from `BitNatRepr` via `BlockFunctions`.

The second part of the calculation is the *main loop*. This is performed for every message block M_i. Its primary purpose is to modify two values X and Y. It uses two other values V and W, and also makes an adjustment to V. The MAA standard specifies it as follows:

```
V := CYC(V);
E := XOR(V, W);
X := XOR(X, Mi);          Y := XOR(Y, Mi);
F := ADD(E, Y);           G := ADD(E, X);
F := OR(F, A);            G := OR(G, B);
F := AND(F, C);           G := AND(G, D);
X := MUL1(X, F);          Y := MUL2A(Y, G).
```

The fifth and sixth lines (OR and AND operations using constants A, B, C and D) are covered in section 4.1.2. They are represented by the operation `FIX1` applied to F and `FIX2` applied to G.

I have tried to make the main loop clearer by splitting it into two parts. The first two lines, above, are concerned with V and W, while the other five are concerned with X and Y. The only connection between these two parts is the value E, calculated in the first part and used in the second.

The second part of the main loop can be treated as a self-contained "core". It begins with a pair of blocks X and Y, and finishes with a pair of modified blocks X and Y. The other inputs are M_i and E. It is represented by an operation

```
MainLoopCore: Pair, Block, Block -> Pair
```

where the arguments are the initial X-Y pair, M_i, and E, respectively. The values of X and Y at different times must be given different names:

```
forall X, Y, Xa, Ya, Xb, Yb, M, E, F, G: Block

    XOR(X, M) = Xa,
    XOR(Y, M) = Ya,
    FIX1(ADD(Ya, E)) = F,
    FIX2(ADD(Xa, E)) = G,
    MUL1(Xa, F)   = Xb,
    MUL2A(Ya, G)  = Yb
  =>
MainLoopCore(Pair(X, Y), M, E) = Pair(Xb, Yb);
```

Being the result of `FIX2`, `G` begins with a zero bit. As explained in section 4.1.3, this means that MUL2A yields the same result as MUL2. `MUL2` could therefore replace `MUL2A` in this conditional equation without changing its meaning. It is intended in practice (and the MAA standard appears to demand) that MUL2A be used, and not MUL2; the above equation suggests this, but only informally. This issue is considered in section 3.1 (a).

This "core" operation can then be used to simplify the definition of the complete main loop. Just as X and Y form a natural pair, so do V and W. Since V is adjusted, the main loop modifies both these pairs, and the modified pairs both form results of the calculation. I have therefore packaged these two pairs as a single object. The only input besides the initial X-Y-V-W combination is M_i. The main loop is represented by an operation:

```
MainLoop: TwoPairs, Block -> TwoPairs
```

The initial pairs are represented by `XY` and `VW`, and the "new" pairs by `XYn` and `VWn`:

```
forall XY, VW, XYn, VWn: Pair, M, V, W, Vn: Block

    VW = Pair(V, W),
    CYC(V) = Vn,
    VWn = Pair(Vn, W),
    MainLoopCore(XY, M, XOR(Vn, W)) = XYn
  =>
MainLoop(TwoPairs(XY, VW), M) = TwoPairs(XYn, VWn);
```

(It may be easiest to read the bottom line before the conditions.)

The main loop is repeated for every block of the message in succession. The effect is to transform the X-Y-V-W combination repeatedly, using successive message blocks as inputs. The overall transformation is represented by an operation:

```
MainLoopRepeated: TwoPairs, Message -> TwoPairs
```

If the message consists of just one block, it can be denoted by `Message(M)`, for some block `M`; and the main loop is performed once. If the message consists of more than one block, it can be denoted by `M + MM`, for some block `M` (the first block) and some message `MM` (the rest of the blocks); the main loop is performed for the first block, and then repeated for the rest of the blocks:

```
forall XYVW, XYVWa: TwoPairs, M: Block, MM: Message

MainLoopRepeated(XYVW, Message(M)) = MainLoop(XYVW, M);
    MainLoop(XYVW, M) = XYVWa
  =>
MainLoopRepeated(XYVW, M + MM) = MainLoopRepeated(XYVWa, MM);
```

The third part of the calculation is the *coda*. In this, the main loop is performed two more times, using the values S and T, in that order, as "message" blocks. Then the final X and Y values are used to calculate Z = XOR(X, Y). So the inputs to the coda are the X-Y-V-W combination it begins with, and a pair of blocks S and T; and the result of the coda is a block Z. The coda is represented by an operation

```
Coda: TwoPairs, Pair -> Block
```

where:

```
forall XYVW, XYVWa, XYVWb: TwoPairs, S, T, Xb, Yb: Block, VWb: Pair

    MainLoop(XYVW, S) = XYVWa,
    MainLoop(XYVWa, T) = XYVWb,
    XYVWb = TwoPairs(Pair(Xb, Yb), VWb)
  =>
Coda(XYVW, Pair(S, T)) = XOR(Xb, Yb);
```

(The final V and W values, represented by `VWb`, are unused.)

As outlined in section 1.2, the MAA ("the algorithm" proper) uses a key and a message to produce a "MAC". Of the values generated by the prelude, X_0, Y_0, V_0 and W form the initial values of X, Y, V and W for the main loop (W is never changed); while S and T are the extra "message" blocks used in the coda. The coda result Z is the "MAC". The MAA is represented by an operation

```
MAA: Pair, Message -> Block
```

where:

```
forall Key, XY, VW, ST: Pair, MM: Message, XYVWn: TwoPairs, Z: Block

    Prelude(Key) = ThreePairs(XY, VW, ST),
    MainLoopRepeated(TwoPairs(XY, VW), MM) = XYVWn,
    Coda(XYVWn, ST) = Z
  =>
MAA(Key, MM) = Z;
```

Note that `MAA` is defined (though not used) for messages of unlimited length.

4.4 Correct use of "the algorithm"

The use of "the algorithm" (the MAA) is to authenticate a message. But using it *correctly* involves two things: "the algorithm" must be applied according to the mode of operation specified in the MAA standard; and it must not be used to authenticate a message with more than 1 000 000 blocks.

4.4.1 Segmented messages

As mentioned in section 1.2, a message is divided into *segments*. All segments except the last are 256 blocks long, while the last segment is from 1 to 256 blocks long. Each segment is a string of blocks; so it amounts to a "message" in its own right, and can be represented by a value of the sort `Message`.

To represent all the segments making up the (complete) message, one can form a *string of segments*; the individual segments can then be identified as the elements of that string, each of which is a "message" in the sense already established. This "higher-level" string embodies both the *contents* of the complete message, and its segmented *structure*; I call it a *segmented message.*

The fifth data type for MAA defines the concept of a segmented message. This is simply a string of "ordinary" messages; so the segments can have any mixture of lengths. For the MAA, the segments must have the specific lengths mentioned above; a segmented message matching this special pattern will be called a *normal* segmented message. It would be more difficult to construct a sort consisting of normal segmented messages only, but this is not necessary.

The new type, called `SegmentedMessage`, is another actualization of the standard library type `NonEmptyString`:

```
type SegmentedMessage is

    NonEmptyString actualizedby Message using

        sortnames
            Bool for FBool
            Message for Element
            SegmentedMessage for NonEmptyString

        opnnames
            Segment for String

endtype
```

The main parameter, `Element`, is defined this time to be the sort `Message`, imported from *type* `Message`. That is, the elements of the new strings will be messages. The operation parameters `eq` and `ne`, associated with these messages, are those defined for strings in type `NonEmptyString`, and inherited by type `Message` when `NonEmptyString` was *first* actualized. The technical parameters `FBool`, `true` and `not` are again defined to be the `Bool`, `true` and `not` imported from `Boolean` via `Message`.

The sort `NonEmptyString`, defined in *type* `NonEmptyString` (as being actualized now), is renamed `SegmentedMessage`. So `SegmentedMessage` consists of all strings of one or more messages, ie all segmented messages. The operation `String`, which generates strings of one element, is renamed `Segment`. So the segmented message consisting of just the message (ie segment) `S` is denoted by `Segment(S)`. All other operations defined by type `NonEmptyString` keep the same names in type `SegmentedMessage`.

4.4.2 Preliminaries for describing correct use of "the algorithm"

The mode of operation, and the message length restriction, are both described in the final data type for MAA, called `AppliedMAA`. This imports type `BasicMAA`, since "the algorithm" is used, and type `SegmentedMessage`, since the mode of operation uses segmented messages. The standard library type `DecNatRepr` is also imported, so that the important numbers 256 and 1 000 000 can be defined conveniently.

I have defined constants

```
256, 1000000: -> Nat
```

using their decimal representations:

```
256 = NatNum(Dec(2) + 5 + 6);

1000000 = NatNum(Dec(1) + 0 + 0 + 0 + 0 + 0 + 0);
```

4.4.3 The mode of operation

In its essence, a message to be authenticated is "flat"; that is, it has no segmented structure. The segmented structure is introduced for calculation purposes, in the mode of operation.

An operation

```
Flatten: SegmentedMessage -> Message
```

gives the "flat" version of a segmented message. If there is just one segment, the segmented message can be denoted by `Segment(S)`, for some "message" `S`; and the "flat" version is simply `S`. If there is more than one segment, the segmented message can be denoted by `S + SS`, for some "message" `S` (the first segment) and some segmented message `SS` (the rest of the segments); the "flat" version can be formed by concatenating the segments:

```
forall S: Message, SS: SegmentedMessage

Flatten(Segment(S))  = S;
Flatten(S + SS)      = S ++ Flatten(SS);
```

Using an operation

```
Normal: SegmentedMessage -> Bool
```

`Normal(X)` gives the status of the condition "`X` is a normal segmented message". If there is just one segment, the segmented message is normal when the segment is from 1 to 256 blocks long. If there is more than one segment, the segmented message is normal when the first segment is 256 blocks long and the rest of the segments match the "normal" pattern:

```
forall S: Message, SS: SegmentedMessage

Normal(Segment(S)) = Length(S) le 256;
Normal(S + SS)     = (Length(S) eq 256) and Normal(SS);
```

When a message has been segmented, "the algorithm" is applied to each segment. The "MAC" obtained for each segment, except the last, is prefixed to the next segment, before calculating the "MAC" for that next segment; and the "MAC" obtained for the last segment is the "actual" MAC for the whole message. For any key and segmented message, the "actual" MAC is given by an operation:

```
MAC: Pair, SegmentedMessage -> Block
```

If there is more than one segment, the "MAC" prefixed to the last segment is what would be the "actual" MAC for all except the last segment:

```
forall Key: Pair, S: Message, SS: SegmentedMessage

MAC(Key, Segment(S)) = MAA(Key, S);
MAC(Key, SS + S)     = MAA(Key, MAC(Key, SS) + S);
```

(In the second equation, the + on the left appends a segment to a segmented message, while the + on the right prefixes a block to a message.) Note that `MAC` is defined (though not used) for segmented messages which are not normal.

The mode of operation is embodied in an operation

```
MAC: Pair, Message -> Block
```

which gives the "actual" MAC for any key and ("flat") message. The correct segmented message is fixed by two conditions: it must have the same contents as the "flat" message; and it must be normal:

```
forall Key: Pair, MM: Message, SS: SegmentedMessage

    MM = Flatten(SS),
    Normal(SS)
  =>
MAC(Key, MM) = MAC(Key, SS);
```

Note that (this second) `MAC` is defined (though not used) for messages of unlimited length.

4.4.4 The message length restriction

The message authenticated must have 1 000 000 blocks or fewer; I call this an *acceptable* message. The sort `AcceptableMessage` consists of all acceptable messages. The method of generating these is untidy; I have used a constructor operation

```
Restrict: Message -> AcceptableMessage
```

where, for each "acceptable" Message value MM, Restrict creates a "shadow" value Restrict(MM) representing MM, while, for each "unacceptable" MM, Restrict(MM) is defined as an arbitrary acceptable message. The arbitrary acceptable message I have chosen is the single block consisting of zeros:

```
forall MM: Message, Zero: Block

    Length(MM) gt 1000000,
    NatNum(Zero) = 0 of Nat
  =>
Restrict(MM) = Restrict(Message(Zero));
```

The phrase "of Nat", in the second condition, is needed because 0 is an overloaded operator: there are constants named 0 in the sorts Nat, Bit and DecDigit; "0 of Nat" identifies the first of these. (The rules which deduce the meaning of an overloaded operator from its context do not make use of the opposite side of an equation.) The second condition fixes the variable Zero; it does not matter what Block value is actually used, but it must still be specific, to ensure that Restrict(MM) is not defined as more than one arbitrary acceptable message.

Each AcceptableMessage value is meant to be regarded as Restrict(MM), where MM is acceptable; it is MM that really embodies the contents of the message. But there is one AcceptableMessage value which is also Restrict(MM) for all unacceptable MM. An operation

```
Contents: AcceptableMessage -> Message
```

identifies which message is "really" represented by any AcceptableMessage value:

```
forall MM: Message

    Length(MM) le 1000000
  =>
Contents(Restrict(MM)) = MM;
```

Correct authentication is represented by an operation

```
Authenticator: Pair, AcceptableMessage -> Block
```

where Authenticator(Key, X) gives the MAC for that key and message:

```
forall Key: Pair, X: AcceptableMessage

Authenticator(Key, X) = MAC(Key, Contents(X));
```

5 Problems with using LOTOS for data security standards

5.1 The lack of partial functions in LOTOS

5.1.1 Partial functions and why they are desirable

The distinction between total and partial functions was mentioned in section 2.1.1, which explained that an operation in LOTOS is always a total function.

The significance of this is illustrated by the operation

```
MAC: Pair, Message -> Block
```

(the second `MAC`) in section 4.4.3. As pointed out at the end of that section, this is *defined* for messages of unlimited length, although it must be *used* only for messages of 1 000 000 blocks or fewer. Ideally, it should be *defined* only for messages of 1 000 000 blocks or fewer.

The issue here is what combinations of argument values are valid. LOTOS uses a simple principle: the valid combinations are those which match the argument sorts listed in the functionality. In practice, one would like to add other conditions. In effect, the line

```
MAC: Pair, Message -> Block
```

states that if `Key` is a `Pair` value and `X` is a `Message` value, then `MAC(Key, X)` is a `Block` value. But one would like to state that if `Key` is a `Pair` value and `X` is a `Message` value, *and* the length of `X` is less than or equal to 1 000 000, then `MAC(Key, X)` is a `Block` value; if the length of `X` is greater than 1 000 000, `MAC(Key, X)` should be undefined. `MAC` would then be a partial function.

In this example, the desired extra condition concerns only one argument. Consequently, an alternative approach would be to define a *subset* of the relevant sort, and specify *that* as the argument sort. In this example, the subset would contain those `Message` values whose length is less than or equal to 1 000 000, and might be called `AcceptableMessage`. This differs from `AcceptableMessage` in section 4.4.4 in that it contains actual `Message` values, not "shadow" values which need explicit conversion to and from `Message` values. So one might write

```
MAC: Pair, AcceptableMessage -> Block
```

where:

```
forall Key: Pair, MM: AcceptableMessage, SS: SegmentedMessage

    MM = Flatten(SS),
    Normal(SS)
  =>
MAC(Key, MM) = MAC(Key, SS);
```

`MM` would still be a `Message` value, and the result sort of `Flatten` would still be `Message`.

This has illustrated two valuable techniques, which LOTOS does not provide: arbitrary *preconditions* (argument restrictions) for an operation, and *subsorts*. With either technique, the material in section 4.4.4, which is the awkward part of my MAA description, could be eliminated. While both techniques are suitable in this example, other situations call for one of them in particular. Preconditions are more natural and convenient if there is some *interdependence* between two or more arguments of an operation; the alternative is to replace these arguments by a single object which embodies them, and define a subsort of such "aggregates". Subsorts are more natural and convenient if a special class of some object is used frequently; the alternative is to state the same precondition every time.

The MAA standard creates little need for partial functions. The main units of calculation (CYC, MUL1 etc; BYT and PAT; the prelude, main loop and coda; and "the algorithm") are total functions by nature. At a more detailed level, some of the steps use partial functions, in effect, but I have defined these steps implicitly rather than as LOTOS operations. For example, the arithmetic functions in section 4.1.3 involve converting blocks into numbers, performing calculations, and then converting numbers back into blocks; converting a number to a block (or to a 33-bit or 64-bit string) is a partial function, because it cannot be done for numbers that are too large. The ability to define partial functions is needed only for the message length restriction, as already explained.

With other data security standards, the need for partial functions may easily cause far more severe problems. For example, schemes based on number theory can be expected to abound in special conditions restricting the numbers involved. The MAA itself provides several clues to how problems could arise; the rest of this section considers these.

I have just mentioned the use of *implicit* calculation steps at a detailed level, to avoid defining LOTOS operations which would represent partial functions, eg converting a number to a block. I have achieved this by stating "reverse" conditions, eg defining a block by stating which number *it* can be converted into. It is precisely implicit conditions of this kind that frustrate automatic analysis by a LOTOS tool, as mentioned in section 3.4. So an MAA description designed to be checked by a tool would contain a number of operations representing partial functions.

The limitations of LOTOS have influenced the definition of blocks. The sort `Block`, in section 4.1.1, uses a constructor operation with 32 arguments. This approach would not be so convenient if blocks were 512 bits long, which is not unrealistic in data security. It would be better if `Block` could be defined as a subsort of `BitString`, by placing a condition on the length. Then the operations `Block`, `BitString`, `eq`, `ne`, `NatNum`, `+` and `++`, defined in sections 4.1.1 and 4.1.3, could be omitted, and the standard operations of `BitNatRepr` used instead. Similar benefits would result from defining `Octet` as a subsort of `BitString`. An operation like XOR could be defined recursively, with two `BitString` arguments, if one could set the precondition that these have the same length.

Section 4.4.1 mentioned the difficulty of constructing a sort consisting of *normal* segmented messages. This would be naturally defined as a subsort of `SegmentedMessage`. It could be used for the "message" argument to the first `MAC` operation in section 4.4.3, to reflect the actual use of this operation. This is not important, because it is easy to define `MAC` for *all* segmented messages. However, an analogous problem which arises with another algorithm-related standard is less tractable. ISO 8372 specifies modes of operation for a 64-bit block cipher. In some modes of operation, the message is made up of blocks whose size is a *parameter*: it ranges from 1 to 64, but is the same for all blocks in a given message. Defining 64 different message sorts, with 64 separate sets of constructor operations, is hardly practical. It is best to represent a block by a bit string, and a message by a string of bit strings. The problem is that this generates messages with mixed block sizes, to which the modes of operation do not apply. Size matching problems also affect the calculations within the modes of operation. This is a case of "restricted flexibility", which calls for partial functions.

Consider a different approach to the question of *message padding*, considered in section 3.1 (b). One might wish to define a `MAC` operation with *three* arguments - the key, the message as a bit string, and a string of padding bits - and assume that padding is to the right of the last block. This would need three preconditions:

(a) The length of the message string must be less than or equal to 32 000 000.

(b) The length of the padding string must be less than 32.

(c) The sum of the lengths of the message and padding strings must be a multiple of 32.

Precondition (c) is another kind of size matching problem.

5.1.2 Circumventing the lack of partial functions

This section considers several ways of trying to circumvent the inability to define a partial function in LOTOS. These are illustrated using the initial example in section 5.1.1, concerning the operation:

```
MAC: Pair, Message -> Block
```

Note that LOTOS provides the ability to state conditions, as can be seen in conditional equations. It is simply that conditions cannot be used as preconditions for an operation, or to define a subsort.

(a) One approach is what I have done in section 4.4.4: where one would like to define a subsort, one creates instead a separate set of "shadow" values. This is constructed by an operation which "converts" the wanted values of the parent sort to their "shadows", while mapping the unwanted values to an arbitrary wanted value.

This has two disadvantages. One is that the "shadow" values are artificially separated from the parent sort, and have to be explicitly converted to and from it. The other is the meaningless connection between the unwanted values of the parent sort and an arbitrary value of the "shadow" sort. If one generates a `Message` value `X`, using "ordinary" operations, one must use the indirect formula

```
Authenticator(Key, Restrict(X))
```

for the MAC. But this is open to "abuse", because one can generate a message `X` with more than 1 000 000 blocks, and use the same formula; this will give the MAC for `Message(Zero)` (where `Zero` is the block consisting of zeros), which is a corruption of the scheme.

Correct application depends on the informal understanding that (using the example) an `AcceptableMessage` value `X` represents `Contents(X)` and no other `Message` value.

(b) A simpler idea is to think of an operation as a partial function and treat it accordingly. One would write equations to define the result for "valid" arguments, and do nothing about "invalid" ones. For example, one would write:

```
forall Key: Pair, MM: Message, SS: SegmentedMessage

    Length(MM) le 1000000,
    MM = Flatten(SS),
    Normal(SS)
  =>
MAC(Key, MM) = MAC(Key, SS);
```

(c) Another idea is to define a special "error" value, and use this as the result when the arguments are "invalid". For example, one would introduce an extra constructor operation

```
Error: -> Block
```

and use the equation in (b) together with:

```
    Length(MM) gt 1000000
  =>
MAC(Key, MM) = Error;
```

These ideas both suffer from a fatal problem.

Consider (c) first. The problem is that whenever an operation uses a `Block` argument, `Error` becomes a possible value. For example, the operation `Pair` creates a multitude of extra pairs of blocks, where one or both is `Error`. Moreover, any of these can form the argument to `PAT`; then the conditions of the equation that defines `PAT` cannot be met, and so in each case `PAT` creates a new `Octet` value. Moreover, the standard library type `Octet` provides operations to extract the eight individual bits from an octet; for the new octets, these operations create a host of new `Bit` values. These new `Bit` values can be combined, 32 at a time, into new `Block` values. And each of these new `Block` values is like `Error`, triggering its own avalanche of new values. New messages can be built using the new blocks, and can be authenticated, generating yet more blocks.

The outcome is an endless explosion of meaningless values. These can be manipulated within a rich and rigorously defined algebra. This does not reflect the intended authentication scheme.

In (b), the consequence of not defining the result for "invalid" arguments is that `MAC` creates a *new* `Block` value in every such case. Each of these new `Block` values has the same effect as `Error` in (c).

(d) In a modification of (c), when any operation acts on an "error" argument, the result is defined as another error. This prevents operations from creating meaningless values. With this approach, each sort must have a special "error" value of its own.

This has several disadvantages. The "error" values are not meant to exist in practice; if behaviour specifications are written, using variable data, "error" values may have to be explicitly forbidden.

The data types must include many extra equations to cover all cases of "error" arguments, as well as conditions to guard equations that do *not* apply to "error" values. All the error-handling parts of the specification are a distraction from the main purpose. The standard library types cannot be used, because they do not provide "error" values.

With a description like that of the MAA, using data types only, correct application depends on the informal understanding that "error" values are not to be implemented and manipulated in practice.

(e) Another modification of (c) creates a separate sort for the result, with "shadow" values representing the "valid" results, plus an "error" value. For example, one can define a sort `MACValue`, with constructor operations:

```
MACValue: Block -> MACValue
Error:      -> MACValue
```

Then one can define

```
MAC: Pair, Message -> MACValue
```

using the equations in (b) and (c), with `MAC(Key, SS)` (at the end of the first equation) replaced by `MACValue(MAC(Key, SS))`. This time, operations do not create meaningless values because their arguments are `Block` rather than `MACValue`, and so cannot be `Error`.

This has the disadvantage that the MAC is not represented by a `Block` value, although in reality it is a block. In some applications, one may wish to manipulate the MAC as a block. One could define an operation which "converts" a `MACValue` value to a `Block` value, and maps `Error` to an arbitrary `Block` value, eg `Zero`. Then the disadvantages would correspond to those of (a); one would use a formula like

```
Block(MAC(Key, X))
```

for the MAC, and this would be open to "abuse".

Correct application depends on the informal understanding that (using the example) `MACValue(x)` represents `x`, while `Error` need not be implemented.

(f) A simple approach is to define a total function, which gives the correct result for "valid" arguments, and an arbitrary result for "invalid" arguments; the intended use of the operation can be stated *informally*. For example, one can simply omit the material in section 4.4.4, and use the total function:

```
MAC: Pair, Message -> Block
```

The informal commentary can mention that the message must not have more than 1 000 000 blocks.

This is clearly the right approach when the function will be used only to define other functions in the specification; the "invalid" arguments will simply not arise. This is the case with the *first* `MAC` operation in section 4.4.3; segmented messages which are not normal are "invalid", but never used.

There are two possible disadvantages. The obvious one, when the function is not used merely to define other functions, is that the goal of expressing everything in formal language is not fully achieved. The other arises when the result for "valid" arguments cannot be naturally extended to "invalid" ones, as it can with both MAC operations; then it becomes necessary to write an artificial extra definition. The final example in section 5.1.1 illustrates this: when the sum of the lengths of the message and padding strings is not a multiple of 32, the definition of the MAC must be specially contrived.

Correct application depends on the informal knowledge of "valid" and "invalid" arguments.

(g) An extension of (f) defines a "companion" function with the same argument sorts and a `Bool` result. The second operation gives `true` for the "valid" arguments and `false` for the "invalid" ones. For example, MAC can be accompanied by an operation

```
MACValid: Pair, Message -> Bool
```

where:

```
forall Key: Pair, MM: Message

MACValid(Key, MM) = Length(MM) le 1000000;
```

This has the disadvantage that MAC still yields a result when the arguments are "invalid", and so is open to "abuse". The second disadvantage of (f) can also arise.

Correct application depends on the informal understanding that the "companion" function determines the applicability of the main function, and need not be implemented in its own right.

(h) A development of (g) combines the two functions, by defining a single object which contains both results. For example, one can define a sort `Authentication`, with a constructor operation

```
Authentication: Bool, Block -> Authentication
```

where `Authentication(true, x)` represents a valid authentication with MAC value `x`, while `Authentication(false, x)` represents an invalid authentication:

```
forall Key: Pair, MM: Message, SS: SegmentedMessage

    MM = Flatten(SS),
    Normal(SS)
  =>
Authenticate(Key, MM) =
    Authentication(Length(MM) le 1000000, MAC(Key, SS));
```

In principle, this is the same as (e), with `Authentication(true, x)` replacing `MACValue(x)`, and all `Authentication(false, x)` values replacing `Error`. Where one then defined an operation to "convert" a `MACValue` value to a `Block` value, one now defines an operation to extract the second component of an `Authentication` value. In essence, the disadvantages of (e) apply here; the second disadvantage of (f) can also arise.

Correct application depends on the informal interpretation of the two components of the result, in analogy with (g).

(i) An extension of (f) uses a behaviour specification to establish the correct use of the operation. For example, one can write:

```
authenticate ? Key: Pair ? X: Message ? C: Block
    [(Length(X) le 1000000) and (C eq MAC(Key, X))]
```

This specifies an observable *event*. Section 2.1.4 described how LOTOS defines a data value abstractly, as a collection of ground terms. LOTOS defines an observable event just as abstractly: the event is defined as a *label* (an identifier) with a *string of data values*. One possible intuitive meaning is that the data values represent actual data handled during the event. In this example, the event has the label `authenticate` and involves three data values. The question marks mean that the values are variable; they are given the names `Key`, `X` and `C`. It is specified that they are of sorts `Pair`, `Message` and `Block`, respectively; intuitively, they represent a key, message and MAC. The square brackets contain a constraint - a condition which must be met. The condition stated is the one that defines a correct authentication. (A "correct authentication" uses any key, involves an "acceptable" message, and generates the "right" MAC for that key and message.) This event specification defines a wide *choice* of observable events, differing in the three data values in the string; the possible events correspond to the set of correct authentications.

This is clearly the right approach in a normal LOTOS specification, where operations are defined only for use in behaviour specifications; "invalid" arguments are simply not used.

With a description like that of the MAA, this approach has the disadvantage that it implies that an event takes place. To have a formal meaning, the above lines must be placed in a proper specification, along with the data types for MAA. This specification will define the *behaviour* of a system. This could be a symbolic system which executes one event and then stops, but this is artificial. One can readily specify an "MAA machine", eg with separate events for loading keys and authenticating messages, but this involves implementation-specific details. The MAA standard defines an idea, not a system.

The second disadvantage of (f) can also arise (even in a normal specification).

Correct application depends on the informal understanding that (using the example) there is one possible event for every correct authentication, but the symbolic event need not correspond to an "actual" event occurring in time.

In conclusion, it can be seen that there are ways of circumventing the lack of partial functions in LOTOS, but they are all unsatisfactory and rely on some informal understanding.

5.2 Standards using an unspecified data algebra

5.2.1 Explanation of the problem

An important feature of the MAA standard is that it is completely specific and self-contained (given my decision in section 3.1 (b)). It specifies that the message is any string of 1 to 1 000 000 blocks, while the key is any pair of blocks, where a block is a string of 32 bits; and for every such message and key, it gives a recipe to identify precisely which one of the 2^{32} possible block values is the right MAC. So the contents of the sorts involved, and the connections between argument and result values of the operation `Authenticator`, are known exactly.

The material in the MAA standard could be embedded within a more comprehensive and application-oriented standard. This standard might define a more complex *algorithmic* scheme, in which forming the MAC is just one element; or it might define a *protocol*, part of which involves transmitting messages together with MAC values. Then one could write a LOTOS description of this more comprehensive standard. My data types could form part of that description, defining the message authentication aspect. The `Authenticator` operation could then be used in expressions, either in the equations of another data type, which define a higher-level operation, or in a behaviour specification, to define the MAC used in some observable event.

Now imagine a more generic version of this standard, which allows users to choose *any* suitable authenticator algorithm. The purpose of this standard is to define the *framework* of a scheme, either algorithmic or a protocol, for achieving some purpose. Message authentication is a necessary part of the scheme, but any good authenticator algorithm will achieve the desired goal, and so the standard does not place a needless restriction on users.

(a) Suppose that a message is still defined as a string of 1 to 1 000 000 blocks, the key as a pair of blocks, and the MAC as a single block, where a block is a string of 32 bits. Then the sorts involved are still completely defined, and can be generated using the same constructor operations as in the MAA description. To represent the authenticator algorithm, one really wants an operation

```
Authenticator: Pair, AcceptableMessage -> Block
```

just as before. The difference now is that this need not be the function defined in the MAA standard, but may be any total function matching this functionality.

What one wishes to specify is that for every `Pair` value `Key` and every `AcceptableMessage` value `X`, `Authenticator(Key, X)` is one of the 2^{32} `Block` values *already created* by the `Block` operation; but one does not wish to specify which one. This cannot be done in LOTOS. Since one does not know which `Block` value `Authenticator(Key, X)` is, one cannot write any equations to define it. But if one writes no such equations, then `Authenticator` creates a *new* `Block` value `Authenticator(Key, X)`. Moreover, this new `Block` value is like the value `Error` in section 5.1.2 (c): it can be used as an argument to operations, leading to an endless explosion of meaningless values.

Another idea is to create a separate sort consisting of MAC values, and write, for example:

```
        Authenticator: Pair, AcceptableMessage -> MACValue
```

`Authenticator` is then the constructor operation for `MACValue`. This represents a higher level of abstraction: one considers the fact that there *is* a MAC value for every key and message, but is no longer concerned with the fact that the MAC is actually a block; then it no longer matters which block it is. This approach does not create new `Block` values, and so avoids the above explosion of meaningless values.

However, one problem remains and another is created:

(i) The new problem is that the MAC may need to be further manipulated in a way which depends on the fact that it *is* a block. For example, the standard may require the message and MAC to be concatenated, and enciphered using 32-bit cipher feedback; the encipherment process would act on a string of *blocks*. One cannot define an operation to "convert" each "abstract" MAC value to its "concrete" block representation, because this raises the same problems as `Authenticator` did previously.

(ii) The existing problem is that the various MAC values, even if they belong to a sort of their own, are not correctly modelled. In practice, many different combinations of key and message must yield the same MAC value. But one does not know *which* combinations yield the same MAC, because this depends on the authenticator algorithm. So one cannot write the equations to make these `Authenticator` results the same. But if one writes no such equations, then the `Authenticator` results are all different. So `MACValue` contains too many values.

To highlight what is wrong, consider the process of checking a message: some communicating entity receives a message and MAC, and compares the MAC with the result of `Authenticator`; if the values differ, the entity takes some special action defined in the standard. This process can be described in a behaviour specification; this would use conditions, like the equality of MAC values, to determine the behaviour. Using the LOTOS semantics, one can then prove that the test always fails (the MAC values are not equal) if the message is not the "right" one; whereas in reality the test might still succeed, because a different message could yield the same MAC. So the specified behaviour is incorrect.

(b) Now suppose, more realistically, that the standard specifies nothing about the nature of the key; that is, the key need not be a pair of blocks, but might be a bit string of some fixed size, or a matrix of numbers satisfying certain properties, or any other structure, depending on the choice of authenticator algorithm. (The standard is also likely to be less precise about the nature of an acceptable message.)

This creates a new problem, because one no longer knows what the "key" sort should contain. The LOTOS semantics determine exactly how many values each sort contains, but this is now unknown. One cannot write the constructor operations to generate the keys, while if one introduces *no* operations that yield key values, there are no keys at all. So besides being unable to provide the right equations, one cannot provide the right operations.

These are hypothetical examples, but the basic problems arise in actual data security standards. One algorithm-related standard, ISO 8372, specifies modes of operation for a 64-bit block cipher; the block cipher is an unspecified function, because the standard applies to *any* 64-bit block cipher; and the nature of the key is unspecified, because it depends on the cipher chosen. In the MAA standard itself, one might take a different approach to the question of *message padding*, considered in section 3.1 (b); as in the VDM and Z descriptions [2, 3], one might wish to use an explicit padding function to transform a bit string into a block sequence, where the contents of the padding field are not specified. Initial indications are that protocol-related data security standards, especially the symbolic protocols mentioned in section 1.1, use algorithmic elements, and indeed simple data items, which are partly unspecified.

In effect, these standards do not define a specific algebra of data values, but allow a choice of algebras. The problem can be viewed as one of *parameterization*: these standards involve a parameterized algebra, with flexibility in the sorts and operations. A LOTOS specification can have parameters, but not of the required kind: the parameters are either *gates*, which are just the labels used to identify observable events, or *variables*, which carry a value selected from some sort and are used in the behaviour specification; they cannot be sorts or operations.

5.2.2 Using parameterized data types

One idea is to use parameterized data types, because these allow sort and operation parameters. Using the hypothetical example in section 5.2.1, the `Authenticator` operation could be a *formal* operation, so that it can represent many different actual operations. The key, message and MAC sorts would have to be formal sorts, because the argument and result sorts of a formal operation must themselves be formal. But this also enables them to represent many different actual sorts, such as different kinds of key.

This technique cannot be used to write a specification of a generic *protocol*. The sorts and operations of a parameterized type are excluded from the final many-sorted algebra, and cannot be used in behaviour specifications. The only use of parameterized types (apart from importing them into other parameterized types) is to actualize them, after which the actualized version (a specific instance) *can* be used in behaviour specifications. The *actual* sorts and operations must come from somewhere inside the same specification; as noted at the end of section 5.2.1, they cannot be parameters to the overall specification. Parameterized types are a mechanism for removing duplication between similar data types, not for leaving choices open to the implementation.

A parameterized type might be used to write a description of a generic *algorithm*. This is because the description, like that of the MAA, may consist of data types only; as explained in section 3.3, this is technically not a specification, but a component of one; and it is more flexible than a specification, in that sort and operation parameters are allowed. For example, a parameterized type could use a formal operation `Authenticator`, representing an unspecified authenticator algorithm, and define, in terms of `Authenticator`, its own (non-formal) operations, representing higher-level algorithmic functions. Then, for any *specific* choice of authenticator algorithm, one could, in principle, write a separate data type to define the authenticator algorithm, and use this to actualize the parameterized type; this actualization would yield an unparameterized type, whose sorts and operations would completely model *that instance* of the standard.

This approach has serious shortcomings. If the description of a standard is based on a parameterized type, it has no formal meaning. The MAA description can be formally interpreted using part of the LOTOS semantics, as described in section 3.3; but the dynamic semantics cannot be applied in the case of a parameterized type, as explained at the end of section 2.3. Instead, the description must be interpreted informally: one must imagine how the parameterized type could be actualized, and then use the formal meaning of the resulting unparameterized type. For this, one must imagine extra type definitions which are "well-behaved", not undermining the description, eg by violating the principles in the last part of section 2.2.2. There may also be "genuine" constraints on the actual parameters; for example, though messages are a formal sort, the standard may still state that they consist of 32-bit blocks; or it may demand that the authenticator algorithm is designed so that messages differing in only one block never yield the same MAC with the same key. The techniques available, such as formal equations, are not powerful enough to express all such requirements.

In short, only an *extension* of the description can have a formal meaning, and what extensions are *valid* is only informally understood. This reflects the fact that LOTOS is being used in a way for which it was not designed. One could say that part of the description is missing, and that this is just a special case of the approach considered in section 5.2.3.

5.2.3 Leaving gaps in the description

A simple approach is to leave a gap in the description wherever information is not available. Wherever an operation is unspecified, one can introduce its name and functionality, but omit the equations defining it. Wherever a sort is unspecified, one can introduce its name, but omit the constructor operations and any equations associated with them. The description may be either a proper specification, to describe a protocol, or a set of data types, to describe an algorithm. Technically, the LOTOS semantics still give the text a formal meaning, because the omissions do not break any syntactic or semantic rules; but this formal meaning must be disregarded, because it is an incorrect model.

Since the text has no *valid* formal meaning, it is not a formal description of the standard. But it may become a formal description when suitably extended - not a formal description of the *standard*, but a formal description of some *implementation*, or *instance*, of the standard, where all choices have been made. In principle, the extension consists of inserting equations or operations in all the gaps, to define the specific operations or sorts chosen. In practice, many auxiliary sorts, operations and types may be needed, like those used to describe the MAA. The additions to the text must be "sensible"; that is, their effect must be to produce sorts and operations of an "intended" nature, without changing the meaning of existing parts of the text in "unintended" ways. As in section 5.2.2, then, only an *extension* of the description can have a useful formal meaning, and what extensions are *valid* is only informally understood.

In other words, one can write a formal description of an *implementation* of the standard, but one writes only an informal description of the standard itself. The standard can be implemented in different ways, each needing a different formal description. The informal description contains the elements common to the different formal descriptions.

Note that interpreting descriptions can involve two kinds of informal understanding. With *any* LOTOS description, there must be some informal interpretation *after* a formal meaning is obtained, because the formal meaning is a mathematical structure consisting of abstract elements, whose intuitive meaning must be understood or explained. With an *informal* LOTOS description, of the above kind, there must also be some informal interpretation *before* a formal meaning is obtained, concerned with how the text can be extended. When informal understanding of the second kind is needed, no formal reasoning can be applied to the description; the description is not self-contained, and has less value.

Writing a description with gaps is still better than using no formal language at all: it does *reduce* the amount of informal explanation, and therefore the scope for misunderstanding. But the mathematical system provided by LOTOS cannot be used, and so one of the important advantages of LOTOS - its completely formal basis - is lost.

5.2.4 Using processes to perform calculations

In defining a calculation involving data values, it is possible to avoid using a LOTOS operation altogether. Instead, one can define a *process*, which specifies the behaviour of a conceptual entity. This entity exists for the purpose of performing data calculations. It may "run" concurrently with other processes in the system, which can communicate with it; when other processes require a calculation to be performed, they send the "input" values to the "calculator" process, and this carries out the calculation and sends the results back.

This technique can be used instead of defining operations *other than constructors*. The values existing in each sort must be created by operations; a process cannot *create* any values, and so the constructor operations are still needed. But other operations always yield values that "already" exist, and a process can also do this. Just as one can define such an operation using equations to specify the result, one can define a process using expressions to specify the "output" values arising from any given "input" values. The behaviour specification defining the actions of the process may resemble a program.

The significance of this approach is that it is possible to specify *non-deterministic* behaviour. One use of non-deterministic specifications is to allow a choice to be made as an implementation design decision. This means that a calculation need not be fully specified, which potentially overcomes the problem in section 5.2.1 (a). The additional problem in section 5.2.1 (b) remains, however, because a process cannot create values in a sort.

This approach is not altogether satisfying. Using a behaviour specification is not as "clean" as using an operation, because it defines not only the result of the calculation, but also a mechanism for obtaining it, involving a succession of events in time. The technique is analogous to that described in section 5.1.2 (i) for describing partial functions, and is an artificial device as far as algorithm-related standards are concerned. If the calculation is "invoked" as one element of a hierarchy of calculations, one is forced to define all higher-level calculations which use this one using the same technique, because a LOTOS operation cannot invoke a process in its definition.

The need to resort to this technique represents a failure of the "abstract data type" part of LOTOS to fulfil its intended role.

There is also an awkward problem of "repeatability". A simple non-deterministic calculation does not have to yield the same result every time it is invoked with the same "input" values. It is possible to specify consistent results by introducing "memory", in the form of variables internal to the process. The extra specification involved in this is cumbersome, and artificial in nature.

Finally, I have already explained that this approach overcomes the problem of unspecified operations but not unspecified sorts. In most, and perhaps all, standards with unspecified operations, there are also unspecified sorts, and so this technique is of no help.

Conclusions

The distinction between algorithm-related and protocol-related standards, in data security, has been considered, because these two classes of standard call for different features in the language used to describe them. LOTOS was designed for describing OSI standards, which suggests it would have the features needed to describe protocol-related standards. On the other hand, a LOTOS specification defines the behaviour of a system in terms of observable events, which is not the purpose of algorithm-related standards. LOTOS might therefore be expected to be suitable for protocol-related, but not algorithm-related, standards in data security.

However, the LOTOS description of the MAA standard, presented in this report, demonstrates that this distinction is not necessarily relevant. Used with care, LOTOS can sometimes yield a clear description of a complex algorithm. This exercise used the "abstract data type" component of LOTOS exclusively; the description consists of a set of data types rather than a proper specification, but it can still be given a formal meaning by using part of the LOTOS semantics.

But the MAA standard has characteristics which make it easier to describe than other standards (given my decision concerning message padding, in section 3.1 (b)). Firstly, it is straightforward, in the sense of being based on *total functions*: the authentication procedure, and its main components, can act on *any* bit strings of the prescribed sizes. Secondly, the procedure is completely specified: the result does not depend on implementation choices. There are other algorithms with these characteristics, such as the American DES (Data Encryption Standard) algorithm; LOTOS may be equally suitable for describing these.

With many algorithms, the inability to define *partial functions* in LOTOS makes the description awkward. The MAA description is not entirely free from this problem, because of a limit on the length of a message. In many cases, the need for partial functions will be more extensive. There are ways of circumventing the limitation, but these can become very cumbersome, and they rely on some informal understanding.

Many standards also leave some aspects unspecified, where choices are left to the implementation. This is a more serious problem, because the LOTOS semantics define a rigid algebra of data values, containing no choice. In certain circumstances, this limitation can be circumvented by using non-deterministic processes, but this is unnatural and awkward. In most, and perhaps all, cases, only an informal LOTOS description can be written, either based on parameterized types or containing gaps. The description itself has no valid formal meaning, but can, in principle, be extended into a description of any *instance* of the standard; the extended text does have a valid formal meaning, but what extensions are valid is only informally understood. No formal reasoning can be applied to the description.

These two problems affect both algorithm-related and protocol-related standards, because the latter (in data security) normally involve the use of algorithms. It is possible that the second problem arises with *all* protocol-related standards in data security.

In the case of algorithm-related standards, both problems are overcome if one uses VDM or Z, because these languages allow partial functions to be defined, and allow some aspects to remain unspecified. So while LOTOS is an effective language for describing a limited selection of algorithm-related standards, all these and many other algorithm-related standards can probably be described well in VDM or Z, which appear to have greater expressive power for this purpose.

In the case of protocol-related standards, there may not yet be a better alternative to LOTOS. This would mean that, in many and possibly all cases, there is no satisfactory language for describing such standards (in data security). If LOTOS is used in these circumstances, the advantage of its completely formal basis is lost, but it still has the advantage of abstract expression. However, it is possible that VDM or Z is adequate for some of these standards, because the standards involve only a limited form of concurrency.

In short, my conclusion is that LOTOS is unsuitable for describing any but a few data security standards, though it may be the *least* unsuitable language for protocol-related ones.

These conclusions apply to LOTOS *in its present form*. The above problems stem from the limitations of the "abstract data type" part of LOTOS. This mechanism for defining data types forms a self-contained language in its own right. It could be replaced by an alternative data type language, without making any change to the rest of LOTOS. If the data type language provided the flexibility of VDM or Z, the resulting combination might form a powerful language for describing both algorithm-related and protocol-related data security standards. But it is possible that *symbolic* protocol standards (mentioned in section 1.1) would still be too abstract: they might define "ideas" or "schemes", rather than "systems" with a prescribed behaviour.

Indeed, the shortcomings of the present data type language result from efforts to limit the timescale in developing LOTOS. Improvements to the language are under consideration, and so a future version of LOTOS may remove the problems I have discussed.

Acknowledgements

I thank Frank Brady, Bernard Chorley and Richard Lampard, all of NPL, for helpful comments on a draft of this report.

This work forms part of NPL's programme on data security, which is sponsored by the Information Technology Division of the Department of Trade and Industry.

References

[1] International Organization for Standardization. *Banking - Approved algorithms for message authentication - Part 2: Message authenticator algorithm.* ISO 8731-2 : 1987 (E).

[2] Parkin G I and O'Neill G. *Specification of the MAA standard in VDM.* NPL Report DITC 160/90, February 1990.

[3] Lai M K F. *A formal interpretation of the MAA standard in Z.* NPL Report DITC 184/91, June 1991.

[4] International Organization for Standardization. *Information processing systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour.* ISO 8807 : 1989 (E).

[5] Van Eijk P H J, Vissers Ch A and Diaz M (editors). *The formal description technique LOTOS.* Elsevier Science Publishers, 1989.

Appendix: Text of the formal description of the MAA standard

```
library

   NonEmptyString, BitNatRepr, Octet, DecNatRepr

endlib
```

```
type BlockFunctions is

    BitNatRepr

    sorts Block

    opns

        _xor_:                 Bit, Bit -> Bit
        2:                     -> Nat

        Block:
                                 Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit,
                                 Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit,
                                 Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit,
                                 Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit
                               ->
                                 Block
        _eq_, _ne_:            Block, Block -> Bool
        BitString:             Block -> BitString
        NatNum:                Block -> Nat

        _+_:                   Bit, Block -> BitString
        _++_:                  Block, Block -> BitString

        CYC:                   Block -> Block
        XOR:                   Block, Block -> Block
        FIX1, FIX2:            Block -> Block
        ADD, MUL1, MUL2, MUL2A: Block, Block -> Block

    eqns

        forall
            x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
            x9,  x10, x11, x12, x13, x14, x15, x16,
            x17, x18, x19, x20, x21, x22, x23, x24,
            x25, x26, x27, x28, x29, x30, x31, x32,
            y1,  y2,  y3,  y4,  y5,  y6,  y7,  y8,
            y9,  y10, y11, y12, y13, y14, y15, y16,
            y17, y18, y19, y20, y21, y22, y23, y24,
            y25, y26, y27, y28, y29, y30, y31, y32: Bit

        ofsort Bit

            0 xor 0 = 0;
            0 xor 1 = 1;
            1 xor 0 = 1;
            1 xor 1 = 0;

        ofsort Nat

            2 = Succ(NatNum(1));
```

```
ofsort BitString

    BitString
      (
        Block
          (
            x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
            x9,  x10, x11, x12, x13, x14, x15, x16,
            x17, x18, x19, x20, x21, x22, x23, x24,
            x25, x26, x27, x28, x29, x30, x31, x32
          )
      )
      =
        Bit(x1) + x2 + x3 + x4 + x5 + x6  + x7  + x8  +
        x9  + x10 + x11 + x12 + x13 + x14 + x15 + x16 +
        x17 + x18 + x19 + x20 + x21 + x22 + x23 + x24 +
        x25 + x26 + x27 + x28 + x29 + x30 + x31 + x32;

ofsort Bool

    forall X, Y: Block

    X eq Y = BitString(X) eq BitString(Y);

    X ne Y = BitString(X) ne BitString(Y);

ofsort Nat

    forall X: Block

    NatNum(X)  = NatNum(BitString(X));

ofsort BitString

    forall X, Y: Block, b: Bit

    b + X = b + BitString(X);

    X ++ Y = BitString(X) ++ BitString(Y);
```

```
ofsort Block

    CYC
      (
        Block
          (
            x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
            x9,  x10, x11, x12, x13, x14, x15, x16,
            x17, x18, x19, x20, x21, x22, x23, x24,
            x25, x26, x27, x28, x29, x30, x31, x32
          )
      )
      =
        Block
          (
            x2,  x3,  x4,  x5,  x6,  x7,  x8,  x9,
            x10, x11, x12, x13, x14, x15, x16, x17,
            x18, x19, x20, x21, x22, x23, x24, x25,
            x26, x27, x28, x29, x30, x31, x32, x1
          );

    XOR
      (
        Block
          (
            x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
            x9,  x10, x11, x12, x13, x14, x15, x16,
            x17, x18, x19, x20, x21, x22, x23, x24,
            x25, x26, x27, x28, x29, x30, x31, x32
          ),
        Block
          (
            y1,  y2,  y3,  y4,  y5,  y6,  y7,  y8,
            y9,  y10, y11, y12, y13, y14, y15, y16,
            y17, y18, y19, y20, y21, y22, y23, y24,
            y25, y26, y27, y28, y29, y30, y31, y32
          )
      )
      =
        Block
          (
            x1  xor y1,  x2  xor y2,  x3  xor y3,  x4  xor y4,
            x5  xor y5,  x6  xor y6,  x7  xor y7,  x8  xor y8,
            x9  xor y9,  x10 xor y10, x11 xor y11, x12 xor y12,
            x13 xor y13, x14 xor y14, x15 xor y15, x16 xor y16,
            x17 xor y17, x18 xor y18, x19 xor y19, x20 xor y20,
            x21 xor y21, x22 xor y22, x23 xor y23, x24 xor y24,
            x25 xor y25, x26 xor y26, x27 xor y27, x28 xor y28,
            x29 xor y29, x30 xor y30, x31 xor y31, x32 xor y32
          );
```

```
FIX1
  (
    Block
      (
        x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
        x9,  x10, x11, x12, x13, x14, x15, x16,
        x17, x18, x19, x20, x21, x22, x23, x24,
        x25, x26, x27, x28, x29, x30, x31, x32
      )
  )
  =
    Block
      (
        x1,  0,   x3,  x4,  x5,  x6,  1,   x8,
        x9,  x10, x11, 0,   x13, 1,   x15, x16,
        0,   x18, x19, x20, 1,   x22, x23, x24,
        x25, x26, 0,   x28, x29, x30, x31, 1
      );

FIX2
  (
    Block
      (
        x1,  x2,  x3,  x4,  x5,  x6,  x7,  x8,
        x9,  x10, x11, x12, x13, x14, x15, x16,
        x17, x18, x19, x20, x21, x22, x23, x24,
        x25, x26, x27, x28, x29, x30, x31, x32
      )
  )
  =
    Block
      (
        0,   x2,  x3,  x4,  x5,  x6,  0,   x8,
        1,   x10, x11, x12, x13, x14, x15, 0,
        x17, 1,   x19, x20, x21, 0,   x23, x24,
        x25, x26, 1,   x28, x29, x30, x31, 1
      );
```

```
        ofsort Block

            forall X, Y, U, L, S, P, D: Block, C, E: Bit

                NatNum(X) + NatNum(Y) = NatNum(C + S)
              =>
            ADD(X, Y) = S;

                NatNum(X) * NatNum(Y) = NatNum(U ++ L),
                NatNum(U) + NatNum(L) = NatNum(C + S),
                NatNum(S) + NatNum(C) = NatNum(P)
              =>
            MUL1(X, Y) = P;

                NatNum(X) * NatNum(Y)                           = NatNum(U ++ L),
                2 * NatNum(U)                                   = NatNum(E + D),
                NatNum(D) + (2 * NatNum(E)) + NatNum(L) = NatNum(C + S),
                NatNum(S) + (2 * NatNum(C))                     = NatNum(P)
              =>
            MUL2(X, Y) = P;

                NatNum(X) * NatNum(Y)         = NatNum(U ++ L),
                2 * NatNum(U)                 = NatNum(E + D),
                NatNum(D) + NatNum(L)         = NatNum(C + S),
                NatNum(S) + (2 * NatNum(C)) = NatNum(P)
              =>
            MUL2A(X, Y) = P;

endtype
```

```
type ConditioningFunctions is

    BlockFunctions, Octet, BitNatRepr

    sorts Pair

    opns

        Pair:                       Block, Block -> Pair

        _xor_:                      Octet, Octet -> Octet
        BitString:                  Octet -> BitString
        NatNum:                     Octet -> Nat

        NeedAdjust:                 Octet -> Bool
        AdjustCode:                 Octet -> Bit
        Adjust:                     Octet, Octet -> Octet

        BYT:                        Pair -> Pair
        PAT:                        Pair -> Octet

    eqns

        ofsort Octet

            forall
                x1, x2, x3, x4, x5, x6, x7, x8,
                y1, y2, y3, y4, y5, y6, y7, y8: Bit

            Octet(x1, x2, x3, x4, x5, x6, x7, x8) xor
            Octet(y1, y2, y3, y4, y5, y6, y7, y8)
              =
                Octet
                  (
                    x1 xor y1, x2 xor y2, x3 xor y3, x4 xor y4,
                    x5 xor y5, x6 xor y6, x7 xor y7, x8 xor y8
                  );

        ofsort BitString

            forall b1, b2, b3, b4, b5, b6, b7, b8: Bit

            BitString(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) =
                Bit(b1) + b2 + b3 + b4 + b5 + b6 + b7 + b8;

        ofsort Nat

            forall B: Octet

            NatNum(B) = NatNum(BitString(B));

        ofsort Bool

            forall B: Octet

            NeedAdjust(B) =
                (B eq Octet(0, 0, 0, 0, 0, 0, 0, 0)) or
                (B eq Octet(1, 1, 1, 1, 1, 1, 1, 1));
```

```
ofsort Bit

    forall B: Octet

    NeedAdjust(B) =>       AdjustCode(B) = 1;
    not(NeedAdjust(B)) => AdjustCode(B) = 0;

ofsort Octet

    forall B, P: Octet

    NeedAdjust(B) =>       Adjust(B, P) = B xor P;
    not(NeedAdjust(B)) => Adjust(B, P) = B;

ofsort Octet

    forall X, Y: Block, B1, B2, B3, B4, B5, B6, B7, B8: Octet

        BitString(X) =
            BitString(B1) ++ BitString(B2) ++
            BitString(B3) ++ BitString(B4),
        BitString(Y) =
            BitString(B5) ++ BitString(B6) ++
            BitString(B7) ++ BitString(B8)
      =>
    PAT(Pair(X, Y)) =
        Octet
          (
            AdjustCode(B1), AdjustCode(B2),
            AdjustCode(B3), AdjustCode(B4),
            AdjustCode(B5), AdjustCode(B6),
            AdjustCode(B7), AdjustCode(B8)
          );
        ofsort Pair

            forall
                X, Y, Xc, Yc: Block,
                B1,  B2,  B3,  B4,  B5,  B6,  B7,  B8,
                Bc1, Bc2, Bc3, Bc4, Bc5, Bc6, Bc7, Bc8: Octet,
                p1,  p2,  p3,  p4,  p5,  p6,  p7,  p8: Bit

                BitString(X) =
                    BitString(B1) ++ BitString(B2) ++
                    BitString(B3) ++ BitString(B4),
                BitString(Y) =
                    BitString(B5) ++ BitString(B6) ++
                    BitString(B7) ++ BitString(B8),
                PAT(Pair(X, Y)) = Octet(p1, p2, p3, p4, p5, p6, p7, p8),
                Adjust(B1, Octet(0,  0,  0,  0,  0,  0,  0,  p1)) = Bc1,
                Adjust(B2, Octet(0,  0,  0,  0,  0,  0,  p1, p2)) = Bc2,
                Adjust(B3, Octet(0,  0,  0,  0,  0,  p1, p2, p3)) = Bc3,
                Adjust(B4, Octet(0,  0,  0,  0,  p1, p2, p3, p4)) = Bc4,
                Adjust(B5, Octet(0,  0,  0,  p1, p2, p3, p4, p5)) = Bc5,
                Adjust(B6, Octet(0,  0,  p1, p2, p3, p4, p5, p6)) = Bc6,
                Adjust(B7, Octet(0,  p1, p2, p3, p4 ,p5, p6, p7)) = Bc7,
                Adjust(B8, Octet(p1, p2, p3, p4, p5, p6, p7, p8)) = Bc8,
                BitString(Xc) =
                    BitString(Bc1) ++ BitString(Bc2) ++
                    BitString(Bc3) ++ BitString(Bc4),
                BitString(Yc) =
                    BitString(Bc5) ++ BitString(Bc6) ++
                    BitString(Bc7) ++ BitString(Bc8)
              =>
            BYT(Pair(X, Y)) = Pair(Xc, Yc);

endtype
```

```
type Message is

    NonEmptyString actualizedby BlockFunctions using

        sortnames
            Bool for FBool
            Block for Element
            Message for NonEmptyString

        opnnames
            Message for String

endtype
```

```
type BasicMAA is

    BlockFunctions, ConditioningFunctions, Message

    sorts TwoPairs, ThreePairs

    opns

        TwoPairs:                Pair, Pair -> TwoPairs
        ThreePairs:              Pair, Pair, Pair -> ThreePairs

        Prelude:                 Pair -> ThreePairs
        MainLoopCore:            Pair, Block, Block -> Pair
        MainLoop:                TwoPairs, Block -> TwoPairs
        MainLoopRepeated:        TwoPairs, Message -> TwoPairs
        Coda:                    TwoPairs, Pair -> Block

        MAA:                     Pair, Message -> Block
```

```
eqns

    ofsort ThreePairs

        forall
            Key: Pair,
            P: Octet,
            Jc, Kc, Q,
            J1_2, J1_4, J1_6, J1_8, J2_2, J2_4, J2_6, J2_8,
            K1_2, K1_4, K1_5, K1_7, K1_9, K2_2, K2_4, K2_5, K2_7, K2_9,
            H4, H5, H6, H7, H8, H9: Block

            BYT(Key) = Pair(Jc, Kc),
            PAT(Key) = P,
            (NatNum(1) + NatNum(P)) * (NatNum(1) + NatNum(P)) = NatNum(Q),
            MUL1(Jc, Jc)     = J1_2,
            MUL1(J1_2, J1_2) = J1_4,
            MUL1(J1_2, J1_4) = J1_6,
            MUL1(J1_2, J1_6) = J1_8,
            MUL2(Jc, Jc)     = J2_2,
            MUL2(J2_2, J2_2) = J2_4,
            MUL2(J2_2, J2_4) = J2_6,
            MUL2(J2_2, J2_6) = J2_8,
            XOR(J1_4, J2_4) = H4,
            XOR(J1_6, J2_6) = H6,
            XOR(J1_8, J2_8) = H8,
            MUL1(Kc, Kc)     = K1_2,
            MUL1(K1_2, K1_2) = K1_4,
            MUL1(Kc, K1_4)   = K1_5,
            MUL1(K1_2, K1_5) = K1_7,
            MUL1(K1_2, K1_7) = K1_9,
            MUL2(Kc, Kc)     = K2_2,
            MUL2(K2_2, K2_2) = K2_4,
            MUL2(Kc, K2_4)   = K2_5,
            MUL2(K2_2, K2_5) = K2_7,
            MUL2(K2_2, K2_7) = K2_9,
            MUL2(XOR(K1_5, K2_5), Q) = H5,
            XOR(K1_7, K2_7)          = H7,
            XOR(K1_9, K2_9)          = H9
          =>
        Prelude(Key) =
            ThreePairs
              (
                BYT(Pair(H4, H5)),
                BYT(Pair(H6, H7)),
                BYT(Pair(H8, H9))
              );
        ofsort Pair

            forall X, Y, Xa, Ya, Xb, Yb, M, E, F, G: Block

                XOR(X, M) = Xa,
                XOR(Y, M) = Ya,
                FIX1(ADD(Ya, E)) = F,
                FIX2(ADD(Xa, E)) = G,
                MUL1(Xa, F)  = Xb,
                MUL2A(Ya, G) = Yb
              =>
            MainLoopCore(Pair(X, Y), M, E) = Pair(Xb, Yb);

        ofsort TwoPairs

            forall XY, VW, XYn, VWn: Pair, M, V, W, Vn: Block

                VW = Pair(V, W),
                CYC(V) = Vn,
                VWn = Pair(Vn, W),
                MainLoopCore(XY, M, XOR(Vn, W)) = XYn
              =>
            MainLoop(TwoPairs(XY, VW), M) = TwoPairs(XYn, VWn);

        ofsort TwoPairs

            forall XYVW, XYVWa: TwoPairs, M: Block, MM: Message

            MainLoopRepeated(XYVW, Message(M)) = MainLoop(XYVW, M);
                MainLoop(XYVW, M) = XYVWa
              =>
            MainLoopRepeated(XYVW, M + MM) = MainLoopRepeated(XYVWa, MM);

        ofsort Block

            forall XYVW, XYVWa, XYVWb: TwoPairs, S, T, Xb, Yb: Block, VWb: Pair

                MainLoop(XYVW, S) = XYVWa,
                MainLoop(XYVWa, T) = XYVWb,
                XYVWb = TwoPairs(Pair(Xb, Yb), VWb)
              =>
            Coda(XYVW, Pair(S, T)) = XOR(Xb, Yb);

        ofsort Block

            forall Key, XY, VW, ST: Pair, MM: Message, XYVWn: TwoPairs, Z: Block

                Prelude(Key) = ThreePairs(XY, VW, ST),
                MainLoopRepeated(TwoPairs(XY, VW), MM) = XYVWn,
                Coda(XYVWn, ST) = Z
              =>
            MAA(Key, MM) = Z;

endtype
```

```
type SegmentedMessage is

    NonEmptyString actualizedby Message using

        sortnames
            Bool for FBool
            Message for Element
            SegmentedMessage for NonEmptyString

        opnnames
            Segment for String

endtype
```

```
type AppliedMAA is

    BasicMAA, SegmentedMessage, DecNatRepr

    sorts AcceptableMessage

    opns

        256, 1000000:           -> Nat

        Flatten:                SegmentedMessage -> Message
        Normal:                 SegmentedMessage -> Bool
        MAC:                    Pair, SegmentedMessage -> Block

        MAC:                    Pair, Message -> Block

        Restrict:               Message -> AcceptableMessage
        Contents:               AcceptableMessage -> Message

        Authenticator:          Pair, AcceptableMessage -> Block

    eqns

        ofsort Nat

            256 = NatNum(Dec(2) + 5 + 6);

            1000000 = NatNum(Dec(1) + 0 + 0 + 0 + 0 + 0 + 0);

        ofsort Message

            forall S: Message, SS: SegmentedMessage

            Flatten(Segment(S)) = S;
            Flatten(S + SS)     = S ++ Flatten(SS);

        ofsort Bool

            forall S: Message, SS: SegmentedMessage

            Normal(Segment(S)) = Length(S) le 256;
            Normal(S + SS)     = (Length(S) eq 256) and Normal(SS);

        ofsort Block

            forall Key: Pair, S: Message, SS: SegmentedMessage

            MAC(Key, Segment(S)) = MAA(Key, S);
            MAC(Key, SS + S)     = MAA(Key, MAC(Key, SS) + S);

        ofsort Block

            forall Key: Pair, MM: Message, SS: SegmentedMessage

                MM = Flatten(SS),
                Normal(SS)
              =>
            MAC(Key, MM) = MAC(Key, SS);
        ofsort AcceptableMessage

            forall MM: Message, Zero: Block

                Length(MM) gt 1000000,
                NatNum(Zero) = 0 of Nat
              =>
            Restrict(MM) = Restrict(Message(Zero));

        ofsort Message

            forall MM: Message

                Length(MM) le 1000000
              =>
            Contents(Restrict(MM)) = MM;

        ofsort Block

            forall Key: Pair, X: AcceptableMessage

            Authenticator(Key, X) = MAC(Key, Contents(X));

endtype
```