RAL-93-099

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-93-099

A Model-Oriented Analysis of a Communications Protocol

J C Bicarregui

December 1993

A Model-Oriented Analysis of a Communications Protocol

Juan Bicarregui

Abstract

In [BA92], Bruns and Anderson describe a communications protocol in CCS with value-passing. A data model of the state is given in terms of the type constructors usually found in model-oriented specification. For the agents described, a series of semaphores ensure exclusive access to the state, thus the behaviour can be described as a purely sequential system.

This paper considers some alternative data-models for this system: two abstractions and two reifications of the original specification are given. In particular, strong invariants are used to exclude unreachable values from the state space. The example raises some stylistic questions concerning how much detail, that can be inferred from the invariant, should be left implicit in postconditions.

VDM is used for the development, and the role of the explicit frames of reference in the operation definitions is examined in some detail. The interaction between read and write frames and invariant is studied, as is the manner by which the information in the frames is propagated during refinement. Also examined, is how the use of these frames can be extended and how their use can be combined with operation structuring mechanisms available in other model-oriented methods.

The paper concludes with a discussion of some general questions of methodology raised by the example.

1 Introduction

In [BA92], Bruns and Anderson describe a communications protocol in CCS with value-passing. A data model for the values is given that is, in effect, a model of the state of the device. This model is defined in terms of the type constructors usually found in model-oriented specification but without the use of invariants.

The part of the protocol that is described is a mechanism for manipulating a series of flags indicating the status of some shared-memory buffers. These flags are used to ensure that there is no "data-tearing" as multiple processors simultaneously read and write to the buffers. For the operations that update these flags, semaphores are used to ensure that each operation has uninterrupted access to the flags. Thus this part of the behaviour can be described as a purely sequential system.

This paper considers some alternative data-models for the specification (and reification) of these status flags. In particular, attention is paid to the use of invariants in the data model and frames

of reference in the operations definitions, neither of which are available in the data modelling language of [BA92]. It is argued that these features can play a key role in describing the system in a "natural" fashion and can thus help to deepen our understanding of the model.

VDM [Jones90] is used for the analysis, though some comment is made on the advantages that would arise from some of the structuring mechanisms available in Z [Spivey88] or B [Abrial93]. Familiarity with the basic concepts and notation of VDM is assumed.

The remainder of this first section is an informal description of the application and desired protocol. The second section presents a formal specification of the system at a level of abstraction similar to the "abstract" description of [BA92]. Motivated by an analysis of the invariant of that specification, section three describes two further abstractions that can be made. Section four provides an alternative model of the system that makes it possible to write more useful framing information about the operations. The fifth section extends the model to the "improved" protocol of [BA92] and the next considers the possibilities arising from structured definitions of operations. The last section is a discussion of some of the points arising from the example and raises some general questions of methodology[1].

1.1 The Multiprocessor Shared-Memory Information Exchange (MSMIE)

MSMIE, Multiprocessor Shared-Memory Information Exchange, is a protocol that addresses intra-subsystem communications with

> "several features which make it ideally suited to inter-processor communications in distributed, microprocessor-based nuclear safety systems" [MSMIE].

It has been used in the embedded software of Westinghouse nuclear systems designs.

The protocol uses multiple buffering to ensure that no "data-tearing" occurs as seperate processors communicate via some shared memory. That is, that data is never overwritten by one process whilst it is being read by another. One important requirement is that neither writing nor reading processes should have to wait for a buffer to become available, another is that recent information should be passed, via the buffers, from writers to readers. In the simplification considered in [BA92] it is assumed that information is being passed from a single "slave" processor, to several "master" processors. Thus, there are several reading processors, "masters", but only one writing, "slave", process.

The information exchange is realised by a system with three buffers. Very roughly, at any time, one buffer is available for writing, one for reading and the third is either between a write and a read and hence contains the most recently written information, or between a read and a write and so is idle.

The status of each buffer is recorded by a flag which can take one of four values:

s - "assigned to slave." This buffer is reserved for writing, it may actually be being written at the moment or just marked as available for writing.

n - "newest." This buffer has just been written and contains the latest information. It is not being read at the moment.

[1] This example development is used as the basis of a comparison of the VDM and B notations in [BicRit93].

m - "assigned to master." This buffer is being read by one or more processors.

i - "idle." This buffer is idle, not being read or written and not containing the latest data.

The names of the master processors that are currently reading are also stored in the state.

As mentioned earlier, neither the buffers themselves nor the slave and master processors that actually access the buffers in parallel are modelled here. This analysis concerns only the operations that modify the buffer status flags. In the system as a whole, these operations are protected by a system of semaphores which allow each operation uninterupted access to the state and thus their behaviour is purely sequential.

There are three of these operations:

slave This operation is executed when a write finishes. *slave* sets the status of the buffer that was being written to "newest" thus replacing any other buffer with this status.

acquire This is executed when a read begins. The new reader name (passed as a parameter) is added to the set of readers and status flags are updated as appropriate.

release Executed when a read ends, this removes a reader from the set and updates flags as appropriate.

The details of the behaviour of these operations are quite intricate and their precise description is left to the formal specification in the following section.

It should be noted however that, as it stands, the protocol could have the undesirable property that information flow from slave to master is be held up indefinitely. This possibility is ruled out in the original system [MSMIE] via timing constraints whereas [BA92] suggests an improvement to the protocol (using a fourth buffer) that eliminates the possibility without recourse to timing arguments. This improved protocol is examined in later sections of this paper.

2 A VDM specification of MSMIE

The state in [BA92] is defined as

> "a set of three pairs (a, l) where a is the buffer status, drawn from $\{\mathsf{i}, \mathsf{s}, \mathsf{n}, \mathsf{m}\}$, and l is the buffer identification, drawn from $\{1, 2, 3\}$. The buffers are given as a set rather than a tuple to enable pattern matching rules in the description of the protocol".

The pattern matching rules do indeed give a concise description of the transitions of the system, in particular, the associative, commutative properties of sets are used to good effect in order to avoid much repetitive case analysis. However, the present author found that considerable effort was required to check that the patterns given were exhaustive and that the effects of overlaps between patterns were sensible. This difficulty is exacerbated by the fact that many of the states in the model are actually unreachable but no invariant on the state type is given to exclude them.

The specification given here makes an alternative choice of a sequence of three buffers for the state description. In addition, an invariant is used to exclude unwanted values from the state type.

2.1 The state

Possible values of the status flags are given via an enumerated type; the type of the names of master (reading) processors is deferred.

types

$$Status = \{\mathsf{s}, \mathsf{m}, \mathsf{n}, \mathsf{i}\}$$

$$MName = \text{token}$$

The state is composed of three buffer status flags and a set of the names of the currently reading masters. The invariant captures the fact that only certain states are reachable by the operations. It gives restrictions to the possible combinations of status flags, namely that there is always exactly one buffer assigned to the writing slave; there is at most one currently being read and at most one with newest data that is not being read; and the set of reader names is empty precisely when there is no buffer being read. The initial state assigns one buffer to the slave and records that the other two buffers are idle.

state Σ of
$b : Status^*$ ← why use a sequence and not a set?
$ms : MName\text{-set}$
inv $mk\text{-}\Sigma(b, ms) \triangleq \text{len } b = 3 \wedge$
$count(\mathsf{s}, b) = 1 \wedge$
$count(\mathsf{m}, b) \in \{0, 1\} \wedge$
$count(\mathsf{n}, b) \in \{0, 1\} \wedge$
$(count(\mathsf{m}, b) = 0 \Leftrightarrow ms = \{\})$
init $mk\text{-}\Sigma(b, ms) \triangleq b = [\mathsf{s}, \mathsf{i}, \mathsf{i}] \wedge ms = \{\}$
end

where[1]

$$count : Status \times Status^* \to \mathbb{N}$$

$$count(s, ss) \triangleq \text{len}\,(ss \rhd s)$$

A validation condition on the state

We observe that only four combinations of buffers are allowed by the invariant:

$$\forall mk\text{-}\Sigma(b, ms): \Sigma \cdot \{b(1), b(2), b(3)\}_m \in \{\{\mathsf{s},\mathsf{i},\mathsf{i}\}_m, \{\mathsf{s},\mathsf{i},\mathsf{n}\}_m, \{\mathsf{s},\mathsf{i},\mathsf{m}\}_m, \{\mathsf{s},\mathsf{n},\mathsf{m}\}_m\}$$

[1] Here, range restriction is used on sequences, viewing them as maps from natural numbers to elements.

where we have used $\{\ldots\}_m$ as a notation for bags (multisets), for example $\{\mathsf{s},\mathsf{i},\mathsf{i}\}_m$ is the bag containing one 's' and two 'i's.

Thus the invariant has captured, and brought to the fore, properties that would otherwise have to be deduced by looking in detail at the definitions of the operations. It makes it possible to build quickly our intuition of the workings of the specified machine. We know immediately that there is always one buffer reserved for writing, at most one being read, and at most one with newest data not being read.

2.2 The Operations

Slave

The first operation, *slave*, is executed when a write completes. It reassigns the status of the buffer just written, previously s, to n, thus replacing any other n buffer. It also non-deterministically chooses another available buffer which is to be the new buffer reserved for writing and assigns to it status s.

slave ()
ext wr b : $Status^*$
pre true
post $\forall i \in \{1,2,3\} \cdot$
$(\overleftarrow{b}(i) = \mathsf{s} \Rightarrow b(i) = \mathsf{n}) \wedge$
$(\overleftarrow{b}(i) = \mathsf{m} \Rightarrow b(i) = \mathsf{m})$

The postcondition may, at first sight, seem to be to liberal: what should happen to any buffer that had status n or i?. However, in conjunction with the invariant and the frame, it ensures that no other n buffer remains, that exactly one new s buffer is chosen, and that no new m buffers are added. Thus for example we can write the following validation property for *slave* which can be proved in order to increase confidence in the correctness of the postcondition:

$$\overleftarrow{b}(i) \in \{\mathsf{n},\mathsf{i}\} \Rightarrow b(i) \in \{\mathsf{i},\mathsf{s}\}$$

Note also that all three implications could have been equivalences without changing the operation.

Acquire

The second operation, acquire, is executed when a read is about to start. It adds the new reader's name, passed as a parameter, to the record of active readers and reassigns status flags as necessary.

If there is a buffer currently being read then the new read also begins to read that same buffer and no status change is required. Otherwise the new read starts on the buffer with newest data, status n, and reassigns the status of that buffer to m.

The operation can only be executed in these two situations and this information is recorded in the precondition which requires that there is either a status m or status n buffer. The precondition also records the fact that the operation is only required to function when the new reader is not already in the set of readers.

Note that, in selecting which buffer is to be read, it is not always possible to choose the buffer with newest data. This situation occurs when there are currently buffers with both status m and n, which arises when the data in the n buffer has become available since the start of an ongoing read, that is, when there has been a *slave* since an *acquire* for which there has not yet been a corresponding *release*. In this situation, were the new master to begin reading the n buffer, there would then be two buffers reserved for reading. Consequently, should another *slave* now occur, attempting to preserve this new data would leave no buffer being available for another write to start, thus contradicting one of the fundamental requirements of the protocol: that processors should never have to wait to gain access to buffers. The invariant is designed to prevent this possibility, by insisting that there is always one (and precisely one) buffer with status s.

$$
\begin{array}{l}
acq\ (l\colon MName) \\
\textsf{ext wr } b \quad : \mathit{Status}^* \\
\quad \textsf{wr } ms\ :\ MName\text{-set} \\
\textsf{pre } l \notin ms \wedge \\
\quad \exists i \in \{1,2,3\} \cdot b(i) = \mathsf{n} \vee b(i) = \mathsf{m} \\
\textsf{post } ms = \overleftarrow{ms} \cup \{l\} \wedge \\
\quad \forall i \in \{1,2,3\} \cdot \\
\qquad \textsf{if } \overleftarrow{b}\,(i) = \mathsf{n} \wedge \overleftarrow{ms} = \{\,\} \textsf{ then } b(i) = \mathsf{m} \textsf{ else } b(i) = \overleftarrow{b}\,(i)
\end{array}
$$

It is worth observing that the last line of the postcondition could have been written as

$$\textsf{if } \overleftarrow{b}\,(i) = \mathsf{n} \textsf{ then } b(i) \in \{\mathsf{n},\mathsf{m}\} \textsf{ else } b(i) = \overleftarrow{b}\,(i)$$

or simply as

$$\overleftarrow{b}\,(i) \neq \mathsf{n} \wedge ms \neq \{\,\} \ \Rightarrow\ b(i) = \overleftarrow{b}\,(i).$$

The apparent non-determinism in the alternatives is illusory because the invariant will ensure that there is no real choice as to the status to assign to any buffer that previously had status n. However, the longer and apparently stronger postcondition is preferred as the shorter versions seem to be more cryptic.

Release

The release operation is executed when a reading, master processor finishes its read. The name of the processor is removed from the set of readers and again, status flags reassigned as required.

If this master is not the last one currently reading, then no change is required to the status flags. However, if this is the last master currently reading the m buffer, then this buffer must have its

flag reassigned. There are two possibilities. On the one hand, should there be another buffer with status n available at this time, that is if a write has been completed since the current "chain of reads" began on this buffer, then the m buffer no longer contains the most recent data and so should now be set to i. On the other hand, if there has been no write since the chain of reads began, and hence there is no n buffer available, the m buffer contains the most recent data and its status should be reset to n.

$rel\ (l\colon MName)$
ext wr b : $Status^*$
wr ms : $MName$-set
pre $l \in ms$
post $ms = \overleftarrow{ms} - \{l\} \wedge$
$\forall i \in \{1,2,3\} \cdot$
if $ms = \{\} \wedge \overleftarrow{b}(i) = \mathsf{m}$
then $b(i) \in \{\mathsf{n},\mathsf{i}\} \wedge count(\mathsf{n}, b) = 1$
else $b(i) = \overleftarrow{b}(i)$

Again there is some choice as to how much of the information that is deducible from the invariant should be made explicit in the postcondition. For example the first conjunct of the 'then' clause, $b(i) \in \{\mathsf{n},\mathsf{i}\}$, could have been omitted as no other possibilities are permitted by the invariant, or alternatively, the whole 'then' clause could be replaced by a more explicit form

$$\text{if } \exists \mathsf{j} \in \{1,2,3\} \cdot \overleftarrow{b}(j) = \mathsf{n} \text{ then } b(i) = \mathsf{i} \text{ else } b(i) = \mathsf{n}$$

It is debatable which gives the clearer specification.

2.3 Discussion

Invariants and postconditions

In this presentation, the invariant has been used to convey quickly an understanding of the reachable values of the state. In VDM, the state invariant is effectively part of the state typing information and as such is assumed to be maintained by the operations.

This implicit maintenance of the invariant leads to the choices discussed above of how much of the information deducible from the invariant should be repeated in a postcondition. There is often some tension between the most concise form that relies on properties of the invariant for its correctness, and a longer, but more explicit form, that includes some redundant information. This choice can be seen as an opportunity to prove the stronger forms from the weaker. Which form is chosen may make a significant difference to the complexity of the proofs: the form that most clearly conveys the information may not be the form that will be most usable in proofs. Indeed, the stronger form is more likely to be helpful when the specification is being proved to be a reification of another, and the weaker form when it is itself being reified. By proving one form from the other, one can move some of the burden of proof that otherwise might have arisen when justifying a data reification into the validation of the single specification. This may

well be to some advantage for reasoning in the context of a single specification is likely to be less complex than reasoning about a pair of specifications.

By contrast, in the B notation, one always has to write operations that imply the preservation of the invariant. This may encourage a tendancy to describe *how* the invariant is maintained, and thus to less abstract specifications. The present example is also considered in [BicRit93] where it is used as the vehicle for a comparison of the VDM and B notations.

Methodology

This specification has given a fairly algorithmic description of which buffers are assigned to what status by each operation. This is a good level of abstraction at which to reason about whole system safety properties such as the freshness of the data transferred from slave to masters which is the focus of [BA92]. Much of the detail of this specification is, however, undesirable clutter for other purposes and it is interesting to give more "external" views of the system, as is done in the next section.

3 Two more-abstract specifications

In this section we give two formal abstractions in the above specification. The new specifications maintain the same external behaviour, however the abstract states are progressively simpler than the one just given. The abstractions arise by ignoring detail in the state model that is unecessary to capture the external behaviour. Retrieve functions from concrete to abstract states are also given which are many-to-one, thus demonstrating "implementation-bias" in the concrete specification.

3.1 A first abstraction: ignoring the identity of buffers

Taking inspiration from the validation condition on the state of the above specification, we can give a more abstract specification where the identity of buffers is ignored and only the four possible *combinations* of buffers are distinguished.

A new enumerated type is given that comprises four tokens, each representing one of the possible combinations of buffers

types

$$Status_1 = \{\mathsf{sii}, \mathsf{sin}, \mathsf{sim}, \mathsf{snm}\}$$

The state simply records which combination is current and the invariant and initialisation are the "images under retreival" of the concrete ones

state Σ_1 **of**

$$bs : Status_1$$
$$ms : MName\text{-}\mathbf{set}$$

inv $mk\text{-}\Sigma_1(bs, ms) \triangleq ms = \{\} \Leftrightarrow bs \in \{\mathsf{sii}, \mathsf{sin}\}$

init $mk\text{-}\Sigma_1(bs, ms) \triangleq mk\text{-}\Sigma_1(\mathsf{sii}, \{\})$

end

The operations are similar to those given in the previous specifications, in particular, the postconditions rely on the same case distinctions.

$slave\ ()$
ext wr bs : $Status_1$
rd ms : $MName$-set
pre true
post $(\overleftarrow{bs} \in \{\mathsf{sii}, \mathsf{sin}\} \Rightarrow bs = \mathsf{sin}) \wedge$
$(\overleftarrow{bs} \in \{\mathsf{sim}, \mathsf{snm}\} \Rightarrow bs = \mathsf{snm})$

$acq\ (l{:}\ MName)$
ext wr bs : $Status_1$
wr ms : $MName$-set
pre $l \notin ms \wedge bs \neq \mathsf{sii}$
post $ms = \overleftarrow{ms} \cup \{l\} \wedge$
if $\overleftarrow{ms} = \{\}$ then $bs = \mathsf{sim}$ else $bs = \overleftarrow{bs}$

$rel\ (l{:}\ MName)$
ext wr bs : $Status_1$
wr ms : $MName$-set
pre $l \in ms$
post $ms = \overleftarrow{ms} - \{l\} \wedge$
if $ms = \{\}$ then $bs = \mathsf{sin}$ else $bs = \overleftarrow{bs}$

The retrieve function from the first more concrete specification to this one is simple to define by cases. As it is usual to give more concrete specifications successively higher numbers we will from now on call the state of specification given earlier Σ_2.

$retr_{2\text{-}1} : \Sigma_2 \to \Sigma_1$
$retr_{2\text{-}1}(mk\text{-}\Sigma_2(b, ms)) \triangleq$ cases $(count(\mathsf{n}, b), count(\mathsf{m}, b))$ of
$(0,0) \to mk\text{-}\Sigma_1(\mathsf{sii}, ms)$
$(1,0) \to mk\text{-}\Sigma_1(\mathsf{sin}, ms)$
$(0,1) \to mk\text{-}\Sigma_1(\mathsf{sim}, ms)$
$(1,1) \to mk\text{-}\Sigma_1(\mathsf{snm}, ms)$
end

This specification abstracts away from the behaviour of the individual buffers and exhibits a useful partitioning of the original state space. We can understand much of the behaviour of MSMIE without concern for the finer detail of the more concrete specification. The specifications also makes it clear that after the initial *slave* operation the system never returns to the sii state. This motivates the next abstraction.

3.2 A yet more abstract specification

A further abstraction can be obtained by noticing that the distinction between the initial sii state and the rest of the state space is all that is required to model the external behaviour. Apart from the precondition of *acquire* which effectively requires that a *slave* should occur before any *acquire*, the buffer part of the state is entirely redundant. Thus, the place of the buffers can be taken by a single boolean flag that records whether a *slave* has ever occurred.

state Σ_0 of
b : B
ms : $MName$-set
inv $mk\text{-}\Sigma_0(b, ms)$ $\triangleq$ $b = \text{false} \Rightarrow ms = \{\}$
init $mk\text{-}\Sigma_0(b, ms)$ $\triangleq$ $b = \text{false} \wedge ms = \{\}$
end

The retrieve function is straightforward.

$retr_{1\text{-}0} : \Sigma_1 \to \Sigma_0$
$retr_{1\text{-}0}(mk\text{-}\Sigma_1(bs, ms))$ $\triangleq$ $mk\text{-}\Sigma_0(bs \neq \text{sii}, ms)$

The operations specifications are also simple:

slave ()
ext wr b : B
pre true
post b = true

acq (l: $MName$)
ext rd b : B
wr ms : $MName$-set
pre $b = \text{true} \wedge l \notin ms$
post $ms = \overleftarrow{ms} \cup \{l\}$

rel (l: $MName$)
ext wr ms : $MName$-set
pre $l \in ms$
post $ms = \overleftarrow{ms} - \{l\}$

An alternative state model, equivalent to this, would have comprised of a single component of the optional type [$MName$-set]. In this model, nil represents the state before any write occurred, corresponding to the case where b = false and $ms = \{\}$, and otherwise the state is simply ms.

state Σ_{0_a} of
ms : [$MName$-set]
init $mk\text{-}\Sigma_{0_a}(ms)$ $\triangleq$ ms = nil
end

slave ()
ext wr ms : [*MName*-set]
pre true
post if $\overleftarrow{ms} = \text{nil}$ then $ms = \{\}$ else skip

acq (l: *MName*)
ext wr ms : [*MName*-set]
pre $ms \neq \text{nil} \wedge l \notin ms$
post $ms = \overleftarrow{ms} \cup \{l\}$

rel (l: *MName*)
ext wr ms : [*MName*-set]
pre $ms \neq \text{nil} \wedge l \in ms$
post $ms = \overleftarrow{ms} - \{l\}$

3.3 Discussion

Read frames and union types

Describing the state as an optional type is a special case of using a union type for a state component. In this case, it is tempting to write the externals clause of *acq* and *rel* as of the type without nil, and thus highlight the fact that the system can never return to the initial state. This introduces the more general possibility of partitioning the state space by the use of union types for state components in order to bring into the externals information that would otherwise be part of the postcondition.

In the *acquire* for Σ_0 we saw that read access to b was required even though b was only mentioned in the precondition. Clearly an implementation of *acquire* would not require any access to this state component. This is our first minor encounter with the dual nature of role of the read frames in operation definitions which will be discussed at length later. Here the reader is simply asked to note that, as mentioned above, in the alternative state, Σ_{0_a}, omitting the nil value from the type when it is mentioned in the externals could side-step the matter.

Methodology

This specification captures the 'external' behaviour of the protocol. For example it shows that it is always possible to perform a write and, so long as a write has ever occured, it is always possible to start a new reader and release any existing readers. Such properties are then preserved by the refinement and so give us the validation of the corresponding property of the more concrete specifications. This particular validation condition can easily be given using the closure of the disjunction of the three postconditions: However, a general formalism to prove such "global" properties concerning system behaviour, perhaps with modal operators and quantification over "all operations", is not available for VDM.

4 An alternative view of MSMIE

The above specifications are based on the state recording the status of each buffer; effectively the state is a map from buffer names to their status. Returning to the original specification, we recall that there is always exactly one buffer with status s and at most one with status m or n. This makes it possible to invert the map and think of the state as mapping from each status to a buffer.

This leads to a specification that is equivalent to the first one, but which could yield a more efficient basis for an implementation. This change also makes it possible to specify the read and write access constraints more precisely.

4.1 The state

types

$BName = \{1,2,3\}$

$MName =$ not-yet-defined

state Σ_3 of

$s : BName$
$m : [BName]$
$n : [BName]$
$ms : MName\text{-set}$

inv $mk\text{-}\Sigma_3(s,n,m,ms) \triangleq (\mathsf{m} = nil \Leftrightarrow ms = \{\}) \wedge$
$nil\text{-}or\text{-}different([s,m,n])$ (minor) typecheck error in PVS version

init $mk\text{-}\Sigma_3(s,n,m,ms) \triangleq mk\text{-}\Sigma(1,\mathsf{nil},\mathsf{nil},\{\})$

end

where

$nil\text{-}or\text{-}different : [A]^* \to \mathbf{B}$

$nil\text{-}or\text{-}different(l) \triangleq \forall i \in \mathsf{inds}\, l \cdot l(i) = nil \vee l(i) \notin \mathsf{elems}\,(i \lhd l)$

The retrieve function

The retrieve function is again quite simple:

$retr_{3\text{-}2} : \Sigma_3 \rightarrow \Sigma_2$

$retr_{3\text{-}2}(mk\text{-}\Sigma_3(s, n, m, ms)) \triangleq$
  let b_1, b_2, b_3: $BName$ be s.t.
    $\forall i \in \{1,2,3\} \cdot s = i \Rightarrow b_i = \mathsf{s} \wedge$
      $n = i \Rightarrow b_i = \mathsf{n} \wedge$
      $m = i \Rightarrow b_i = \mathsf{m} \wedge$
      $i \notin \{s, n, m\} \Rightarrow b_i = \mathsf{i}$
  in
  $mk\text{-}\Sigma_2([b_1, b_2, b_3], ms)$

4.2 The Operations

slave

slave ()
ext rd m : $[BName]$
  wr n : $[BName]$
  wr s : $BName$
pre true
post $n = \overleftarrow{s}$

Two simple validation properties for *slave* are worth mentioning as they highlight the use of the frames: m is unchanged as it is read-only, and s is non-deterministically assigned to any value permitted by the invariant.

$m = \overleftarrow{m}$
$s \in BName - \{n, m\}$

The interaction between invariant and externals is interesting. Here, read access to m is required although m is not referred to in the specification. This is because m is linked to s via the invariant and the value of s cannot be fully determined by the post-condition. Thus any implementation will need to read m in order to ascertain what value it is valid to assign to s.

On the other hand, in this example, read access is not required to ms although ms is linked to m, and hence also to s, by the invariant. In general however, in order to ascertain the validity of possible implementations, it may be neccessary to draw information from the variables in the transitive closure of the "linked by invariant" relation. Of course, this set of variables may be quite different from those that appear in the operation specification or those that will be accessed by the implementation.

acquire

$acq\ (l\colon MName)$
ext wr ms : $MName$-set
wr n, m : $[BName]$
pre $l \notin ms \wedge \neg(n = \mathsf{nil} \wedge m = \mathsf{nil})$
post $ms = \overleftarrow{ms} \cup \{l\} \wedge$
$(\overleftarrow{ms} \neq \{\} \Rightarrow m = \overleftarrow{m} \wedge n = \overleftarrow{n}) \wedge$
$(\overleftarrow{ms} = \{\} \Rightarrow m = \overleftarrow{n} \wedge n = \mathsf{nil})$

Interestingly, the last conjunct of this postcondition may be considered redundant. When $\overleftarrow{ms} = \{\}$, then $ms = \{l\}$ and so m must be assigned a non nil value. Now, as read access to s is prohibited, the only buffer that we can be sure is not already in use is that previously assigned to n. So any correct implementation that respected the frames of reference must assign this buffer to m. Then the only remaining possible value for n is nil. However, to hide so much information in the externals clauses seems to be counter-productive.

release

$rel\ (l\colon MName)$
ext wr ms : $MName$-set
wr n, m : $[BName]$
pre $l \in ms$
post $ms = \overleftarrow{ms} - \{l\} \wedge$
$(ms \neq \{\} \Rightarrow m = \overleftarrow{m} \wedge n = \overleftarrow{n}) \wedge$
$(ms = \{\} \wedge n \neq \mathsf{nil} \Rightarrow n = \overleftarrow{n} \wedge m = \mathsf{nil}) \wedge$
$(ms = \{\} \wedge n = \mathsf{nil} \Rightarrow n = \overleftarrow{m} \wedge m = \mathsf{nil})$

In this case the variables of the specification and those of the implementation are the same. Even though s is *not* an independent part of the state, read access to s is not needed. It is possible to prove satisfiability and validity of implementations without knowledge of the value of s.

4.3 Discussion

Read frames and scoping

In this specification we have seen some complex interdependencies between the invariant, externals and postcondition. In *slave*, read access to one component was required although that component was not itself referred to in the specification. In *acquire*, information in the externals could be used to define a highly implicit specification. In *release* however, we saw taht some components were not relevant to the operation defintion even though they were related those mentioned in the predicates.

To clarify some of these issues it is helpful to think of the externals clauses not as giving information about the variables mentioned in the *specification*, but rather to see them as giving

"advanced information" of what access to state variables an *implementation* of that operation can be allowed to make. This distinction separates their semantic role giving information about access to state variables from the syntactic role they play in binding the free variables of the precondition and post-condition.

The "linked by invariant" condition partitions the state components into independent subsets and these parts are the level of granularity at which sense can be made of the operation definition. Thus, in general, they are also the finest partition for which satisfiability and refinement can be sensibly defined.

More examples are given in [Bic93] where the roles of frames and invariants in algorithm refinement is considered more closely, independence is defined, and a framework for algorithm refinement with frames is proposed.

5 The improved MSMIE

As mentioned earlier, Bruns and Anderson observe that, as it stands, the three buffer MSMIE can exhibit an undesirable behaviour. That is, it is possible for a series of overlapping reads, each beginning before the last ends, to lock-out indefinitely the lastest data. They suggest an "improved" protocol that uses a fourth buffer to eliminate this possibility.

Surprisingly, although this new protocol exhibits the same external behaviour as the earlier one, there is no formal refinement relationship between them. To understand why this should be recall that the part of the system modelled does not concern itself with the actual assignment of processors to buffers and does not model the actual transfer of information from slave to masters. Thus, no information about the flags is exported from the system, and all the machinations of the state can be seen as purely an implementation bias in the model contributing nothing to the external behavior of the part of the system modelled.

The four buffer version is, however, a refinement of the most abstract specification given earlier, which gave an unbiased model of the external behaviour. This gives another important reason for considering the abstractions. In particular, validations of the abstract model will carry over to both the three and four buffer versions.

Of course, in this case, it is the internal properties of the model itself that are of interest, as it is these properties that influence the freshness of the data read by the masters. In this respect, the four buffer protocol is indeed better behaved as it would lead to a system where the delay in information transfer is at worst equal to that of the three buffer version.

In the four buffer version of MSMIE, there is also an extra status possible for buffers. o is used to denote a buffer that is still being read but no longer contains most up-to-date information.

Thus

s as before, is a buffer that is reserved for writing

n as before, is a buffer that has latest info but is not being read (waiting for read)

m is a buffer being read, (and the newest such)

o is a buffer being read (but there is also a newer one being read)

ms is the set of masters reading m

os is the set of masters reading o.

New masters are always assigned to the n or the m buffer. m buffers are "demoted" to o status in a way that ensures that the o buffer will periodically become idle. In this way the protocol avoids the "refresh" problems of the three-buffer version. Again detailled descriptions of the mechanisms used achieve is postponed until the formal treatment.

It might help to think of $i \rightarrow s \rightarrow n$ as the write phase of a buffer and $n \rightarrow m \rightarrow o \rightarrow i$ as the read phase. Then MSMIE always has two buffers in write phase and two buffers in read phase.

5.1 The state

types

$BName = \{1,2,3,4\}$

state Σ_4 of

s : $BName$
n : $[BName]$
m : $[BName]$
o : $[BName]$
ms : $MName$-set
os : $MName$-set

inv $mk\text{-}\Sigma_4(s,n,m,o,ms,os) \triangleq$
$(m = nil \Leftrightarrow ms = \{\}) \wedge$
$(o = nil \Leftrightarrow os = \{\}) \wedge$
$(ms \cap os = \{\}) \wedge$
$(nil\text{-}or\text{-}different([s,n,m,o])) \wedge$
$(m = nil \wedge n = nil \Rightarrow o = nil)$

init $mk\text{-}\Sigma_4(s,n,m,o,ms,os) \triangleq mk\text{-}\Sigma(1,\mathsf{nil},\mathsf{nil},\mathsf{nil},\{\},\{\})$

end

The form of the last conjunct in the invariant, which rules out $\{s,o,i,i\}_m$, is the result of the way that readers of m are released which, as in the earlier specifications, ensures that there is always an m or an n buffer remaining.

A validation property for the state

The invariant only allows the following 7 combinations of buffer status:

$$\{s,i,i,i\}_m, \{s,i,i,n\}_m, \{s,i,i,m\}_m, \{s,i,m,n\}_m, \{s,i,m,o\}_m, \{s,i,n,o\}_m, \{s,m,n,o\}_m$$

Retrieve function

As stated, this version is a data refinement of the most abstract model. The retrieve function is straightforward:

$$retr_{4\text{-}0} : \Sigma_4 \rightarrow \Sigma_0$$

$$retr_{4\text{-}0}(mk\text{-}\Sigma_4(s, n, m, o, ms, os)) \triangleq mk\text{-}\Sigma_0(n = \mathsf{nil} \wedge m = \mathsf{nil} \wedge o = \mathsf{nil}, ms \cup os)$$

5.2 The Operations

slave

slave ()
ext rd m, o : $[BName]$
wr n : $[BName]$
wr s : $BName$
pre true
post $n = \overleftarrow{s}$

This time more variables will need to be accessed by the implementation than are mentioned in the predicates. The implementation will require access to m and o in order to be able to set a valid s. This is expressed in the validation condition

$$s \in BName - \{n, m, o\}$$

This access requirement is recorded in the externals even though the pre and postconditions do not mention m and o.

The descriptions of *acquire* and *release* that follow are rather unwieldy. They given by case analyses and as different variables change in the different cases, the operations have to have a wide write access and hence require a lot of clauses saying which variables do not change in that case.

For the time being we introduce an informal shorthand for this, but in the next section give structured definitions of operations in a style after Dijkstra's guarded commands and common to other model oriented methods such as Z's schema conjunction and B's parallel generalised substitutions.

$$Id : A^* \rightarrow Expr$$

$$Id(l) \triangleq \forall i \in \mathsf{inds}\, l \cdot \text{“}l(i) = \overleftarrow{l(i)}\text{”}$$

Note that this function must be considered as an informal meta-notational shorthand rather than a formal function defintion. It is not in fact formally correct to mix such metalinguistic constructs in the object level specification.

acquire

Acquire behaves in a manner very similar to before: beginning the read on either the n or the m buffer as appropriate. The only extra consideration is in the case where there is an n buffer

waiting, an m buffer already being read, but there is no o buffer. In this case, where previously the new read would have been assigned to the m buffer, it is now possible to begin the read on the n buffer, hence the improvment to the freshness of the data exchanged. This is achieved by reassigning the buffer that was already being read to o, and correspondingly, the record of processors reading that buffer, ms, gets moved to os; the new read is started on the buffer that was n, thus making it into a new m, and the new reader is recorded in ms. No more masters will now be assigned to the o buffer thus it will eventually become idle and available to go through the write cycle again.

$acq\ (l\colon MName)$
ext wr ms, os : $MName$-set
wr n, m, o : $[BName]$
pre $l \notin ms \cup os \wedge \neg (n = \mathsf{nil} \wedge m = \mathsf{nil})$
post $(ms \cup os = \overleftarrow{ms} \cup \overleftarrow{os} \cup \{l\}) \wedge$
$(\overleftarrow{m} = \mathsf{nil} \Rightarrow m = \overleftarrow{n} \wedge n = \mathsf{nil} \wedge Id([o, os])) \wedge$
$(\overleftarrow{m} \neq \mathsf{nil} \wedge (\overleftarrow{o} \neq \mathsf{nil} \vee n = \mathsf{nil}) \Rightarrow Id([m, n, o, os])) \wedge$
$(\overleftarrow{m} \neq \mathsf{nil} \wedge \overleftarrow{o} = \mathsf{nil} \wedge n \neq \mathsf{nil}$
$\Rightarrow o = \overleftarrow{m} \wedge m = \overleftarrow{n} \wedge n = \mathsf{nil} \wedge os = \overleftarrow{ms} \wedge ms = \{l\})$

release

Release behaves exactly as before but with an extra case to deal with the case where we release a buffer that is reading the o buffer. When the last o reader releases, o becomes idle.

$rel\ (l\colon MName)$
ext wr ms, os : $MName$-set
wr n, m, o : $[BName]$
pre $l \in ms \cup os$
post $ms \cup os = \overleftarrow{ms} \cup \overleftarrow{os} - \{l\} \wedge$
$(\{l\} \subset \overleftarrow{ms} \Rightarrow Id([m, n, o, os])) \wedge$
$(\{l\} = \overleftarrow{ms} \wedge \overleftarrow{n} = \mathsf{nil} \Rightarrow Id([o, os]) \wedge n = \overleftarrow{m} \wedge m = \mathsf{nil}) \wedge$
$(\{l\} = \overleftarrow{ms} \wedge \overleftarrow{n} \neq \mathsf{nil} \Rightarrow Id([n, o, os]) \wedge m = \mathsf{nil}) \wedge$
$(\{l\} \subset \overleftarrow{os} \Rightarrow Id([m, n, o, ms])) \wedge$
$(\{l\} = \overleftarrow{os} \Rightarrow Id([m, n, ms]) \wedge o = \mathsf{nil})$

Note that, in several places, clauses of acq and rel give more detail than required. For example, in last conjunct $o = \mathsf{nil}$ is redundant because we know $os = \{\ \}$. Similarly, the clause $o = \overleftarrow{o}$ in the penultimate line is also unnecessary. Again, the redundant clauses are left in for the sake of clarity.

Clearly, with this type of operation where the postcondition consits of many cases, the operation definitions become rather difficult to read. A more structured approach to the definition of operations can be taken in Z through the use of the schema calculus, in particular schema conjunction would be particularly useful here[Spivey88]. The B notation also provides a similar structuring mechanism through parallel combination of gernalised substitutions. These however are subject to some syntactic constraints to ensure non-interference which are in general

unnecessarily restrictive. The next section considers how this kind of structuring mechanism could be added to the VDM notation.

6 Structuring the improved MSMIE

With the same state as given above, it is useful to consider how Z style operation conjuction can be used to give a clearer specification of the *acquire* and *release* operations.

This section is more exploratory and introduces some new notation to VDM. Only an informal description of the new constructs is given here, a formal treatment of this style of operation defintion is the subject of ongoing research by the author. In particular the interaction between such structuring mechanisms and the frames of externals is being considered.

Acquire

Acquire is broken into three cases:

$$
\begin{array}{l}
acq\ (l\text{: } MName) \ = \ acq\text{-}to\text{-}m(l) \bigvee acq\text{-}n\text{-}to\text{-}m(l) \bigvee acq\text{-}m\text{-}to\text{-}o(l) \\
\textsf{ext wr } ms, os \ : \ MName\text{-set} \\
\quad \textsf{wr } n, m, o \ : \ [BName] \\
\textsf{pre } l \notin ms \cup os \wedge \neg (n = \textsf{nil} \wedge m = \textsf{nil}) \\
\textsf{post to-be-calculated}
\end{array}
$$

This disjunction of operations is a guarded non-deterministic choice: preconditions are disjoined; postconditions are expanded by difference in frames, guarded by hooked preconditions and then conjoined; externals are unioned but also need to be sufficient to test the guards.

Externals, precondition and postconditions are optional. They can be calculated from the definition and invariant, but it may be useful to give them in cases where a less strict condition than would be calculated is what is required. (That is for wider externals, stronger pre or weaker post.) Giving them explicitly also breaks up the proofs by, in effect, giving a lemma about the specification.

The first sub-operation is called when new reader will be absorbed into an existing set of readers. For this to occur, there must be a buffer m status and also it must be impossible to demote this buffer to o status. This last requirement means that, either there must be no n buffer, or if there is, then there must also be an o buffer. These conditions are captured in the precondition.

$$
\begin{array}{l}
acq\text{-}to\text{-}m\ (l\text{: } MName) \\
\textsf{ext wr } ms \ : \ MName\text{-set} \\
\textsf{pre } m \neq \textsf{nil} \wedge (n \neq nil \ \Rightarrow \ o \neq \textsf{nil}) \\
\textsf{post } ms = \overleftarrow{ms} \cup \{l\}
\end{array}
$$

Although m, n, and o are mentioned in the precondition, the implementation will not require access to any of these so they are not included in the read frame. This shows how the free

variables of the operation pre and postcondition might not be the same as those variables that we are specifying could be be accessed by the implementation.

Note that, as stated here, we have weakened the precondition slightly by not requiring that $l \notin ms$. In fact we know, however, that the operation will only be called in situation where this is indeed that case.

The second sub-operation is called when acquire will cause an n to become an m:

acq-n-to-m (l: *MName*)
ext wr ms : *MName*-set
wr n, m : [*BName*]
pre $n \neq \text{nil} \wedge m = \text{nil}$
post $ms = \overleftarrow{ms} \cup \{l\} \wedge$
$m = \overleftarrow{n} \wedge n = \text{nil}$

The third sub-operation is called when the ms will be demoted to os:

acq-m-to-o (l: *MName*)
ext wr ms, os : *MName*-set
wr n, m, o : [*BName*]
pre $n \neq \text{nil} \wedge m \neq \text{nil} \wedge o = nil$
post $os = \overleftarrow{ms} \wedge ms = \{l\} \wedge$
$o = \overleftarrow{m} \wedge m = \overleftarrow{n} \wedge n = \text{nil}$

Release

Release is broken into four cases:

rel (l: *MName*) = *rel-from-os*(l) $\vee$ *rel-m-to-m*(l) $\vee$ *rel-m-to-i*(l) $\vee$ *rel-m-to-n*(l)
ext wr ms, os : *MName*-set
wr n, m, o : [*BName*]
pre $l \in ms \cup os$
post to-be-calculated

The first sub-operation is called when new reader will be released from os. It handles both the case when os becomes empty and when it does not.

rel-from-os (l: *MName*)
ext wr os : *MName*-set
wr o : [*BName*]
pre $l \in os$
post $os = \overleftarrow{os} - \{l\} \wedge o \in \{\overleftarrow{o}, \text{nil}\}$

Again the last clause could be thought of as redundant: if os becomes empty then the invariant will ensure that o is assigned to nil; however, if os remains nonempty, the only choice that we know will not break the invariant is to leave the o buffer as is.

The second sub-op is called when l will be dropped from ms, but ms remains non-empty:

$rel\text{-}m\text{-}to\text{-}m\ (l\text{: }MName)$
ext wr ms : $MName$-set
pre $\{l\} \subset ms$
post $ms = \overleftarrow{ms} - \{l\}$

The third sub-op is called when l will be dropped from ms causing ms to becomes empty in the presence of an n:

$rel\text{-}m\text{-}to\text{-}nil\ (l\text{: }MName)$
ext wr ms : $MName$-set
wr m : $[BName]$
pre $\{l\} = ms \wedge n \neq$ nil
post $ms = \{\}$

The invariant ensures that $m =$ nil.

Again we have a variable, n, appearing in the precondition that is not in the frame.

Fourth sub-op is called when dropping l from ms will cause ms to becomes empty when no n is present:

$rel\text{-}m\text{-}to\text{-}n\ (l\text{: }MName)$
ext wr ms : $MName$-set
wr m, n : $[BName]$
pre $\{l\} = ms \wedge n =$ nil
post $ms = \{\} \wedge m =$ nil $\wedge\ n = \overleftarrow{m}$

This structuring of the operation has enabled a more progressive defintion of the oerations. It has also afforded the chance to give narrow frames to the sub-operations and has thus precipitated more concise predicates.

These specifications have shown that far from being a minor concern, there are clearly many interesting questions of design choice and methodology that arise from careful consideration of the read and write frames in model-oriented operation specifications.

In VDM operations, the semantic role of the read frame is often underplayed. Typically, it is interpreted as merely providing syntactic scoping for variables appearing in the precondition or postcondition. Alternatively, it could be interpreted as a constraint on implementations - restricting which state components can be read. Thus rather than think of the externals clauses as giving information about the variables mentioned in the *specification*, we see them as giving "advanced information" of what access to state variables the eventual *implementation* of that operation can be allowed to make.

These issues are taken up in more detail in [Bic93] but for the present paper we end with a general discussion of methodological questions that arise from the comparison of this model-oriented treatment of the example given here with the previous analysis using CCS.

7 General Discussion: Methodological Issues

This paper has provided model oriented development of an application that was previous analysed through formalisation in a process algebraic notation (CCS) [BA91, BA92]. As a comparative study it highlights the different benefits of each approach and raises some questions concerning the choice methodogy to be used in the development process. This section gives a brief discussion of some of these concerns.

One of the first questions the system designer must ask him/herself when deciding to undertake the formal development of a software system is which of the broad churches of formalism to adopt. Obviously the answer to this question will depend on many things. In practice, one consideration will be the existing skills of the personnel that will undertake the work, but clearly, the choice should also depend on the type of system that is going to be defined and, in particular, what features of that system are to be the subject of analysis.

Two possible approaches to system description are the so called 'Model-Oriented' and 'Process Algebra' based formalisms. One criterion that may be considered important in choosing between these is the degree of concurrency envisaged for the system, or perhaps more accurately, the degree to which it is the concurrency in the system that will be the subject of the formal analysis to be undertaken. Model-oriented (MO) methods are typically advocated for sequential systems and process-based methods for the study of concurrency.

In process algebra (PA) we know, from the unfolding lemma, that it is always possible to abstract away from parallelism, that is to view the system behaviour from the outside. We may then study whole system behaviours such as equivalences between different combinations of operations. MO methods are not so atuned to studying these behaviours, rather they are typically used to study the system at the operation and data level. That is they are used study the effects of individual operations on the state or alternative possible models for the data.

There is, of course, a direct correspondance between the two views: both define an abstract machine: a state space and the possible transitions between states. The difference is merely one of presentation: MO defines, for each operation, what changes of state are possible; whereas, PA gives, for each state, what operations are possible. Notions of refinement are correspondingly similar.

The different formalisms are best suited to the different concerns that arise at different stages of the development process. Some validation of a formal description is likely to be in terms of the behaviour of the whole system: for example we may wish to assert that we cannot reach certain states, or that we cannot reach a state when no operation can apply. Other validation statements are about individual operations, for example, if some condition holds before an operation then it will also hold afterwards. In PA we have specific calculi, such as Hennessey-Milner logic and its modal extensions for validating whole system properties. The equivalent methodology does not seem to have been worked out for MO methods.

On the other hand, issues surounding data refinement have been largely ignored in PA as the

reduction of the value passing calculus to the pure calculus side-steps such issues. It would seem to be an obvious step forward to develop a formalism that combines the benefits of the two camps.

It is quite simple to imagine a formalism that is a hybrid of the two. One could consider this to be PA with value passing or MO with interleaving. It would describe the overall system behaviour in a style like PA but without parallelism and enable "holistic" validation on this model. Then it would be possible to proceed with MO style data and algorithm refinement. During design process, possible parallelism may re-emerge. Data refinement can lead to the identification of independant parts of the machine that may be implemented separately, algorithm development may make code where temporal ordering of statements is unimportant. Such parallelism could be reintroduced as it arose. In order to do allow such underspecification in algorithm refinement, "framing" the area of influence of each operation is vital in order to control the parallelism and avoid unnecessary bias at each stage. Making best use of the read and write frames may go some way towards this.

References

[Abrial93] The B Method. J.R.Abrial. To be published, 1993.

[Bic93] Bicarregui, J.C. , Algorithm Refinement with Read and Write Frames. Formal Methods Europe '93. LNCS 670, Springer-Verlag.

[BicRit93] Bicarregui, J.C. and Ritchie, B., Invariants, Frames and Postconditions: a comparison of the VDM and B notations. Formal Methods Europe '93. LNCS 670, Springer-Verlag.

[BA91] Bruns, G. and Anderson, S., The Formalization of a Communications Protocol. LFCS TR 91-137 (April 1991).

[BA92] Bruns, G. and Anderson, S., The Formalization of a Communications Protocol. LFCS/Adelard TR. Safety-Critical Computer Systems, April 6, 1992.

[Spivey88] Understanding Z. J.M. Spivey, Cambridge University Press, 1988.

[Jones90] Systematic Software Devlopment using VDM (second edition), C.B. Jones. Prentice Hall, 1990.

[Mor91] Programming from Specifications, C. Morgan, Prentice Hall.

[MSMIE] L.L. Santoline *et al.* Multiprocessor Shared-Memory Information Exchange. IEEE Transactions on Nuclear Science. Vol.36. No.1, Feb 1989. pp. 626-633.