MMAIM: A Software Development Method for Ada
Part II--Example

Jeffrey R. Carter
Senior Engineer, Software
Martin Marietta Astronautics Group
P. O. Box 179
Denver, CO 80201

This work was funded by Martin Marietta Astronautics Group.

ABSTRACT
Part I of this paper described the notation and method for the Martin Marietta Ada Implementation Method (MMAIM). Part II demonstrates the use of MMAIM with an example. This allows the quality of the solution produced by MMAIM to be evaluated.

EXAMPLE: REMOTE DATA ACQUISITION SYSTEM
In this section I demonstrate the use of MMAIM with a version of the Remote Data Acquisition System (RDAS) problem given by Cherry [5]. This problem is sufficiently small for presentation, but adequately complex and concurrent to give a good feel for all of MMAIM's features.

Problem Specification. The purpose of RDAS is to collect data from a spacecraft and transmit them to a ground computer. RDAS will periodically query a multi-channel Analog to Digital Converter (ADC) for the readings from a specified set of channels, and transmit the readings to the ground computer. The ground computer specifies which channels are to be read, and the interval between readings for these channels. Channels which are to be read are called "open" channels; channels not to be read are called "closed."

RDAS will:
1. receive orders (control packets) from the ground computer; these packets contain
    a. the desired channel status (open or closed)
    b. the channel identifier
    c. the interval between readings for the channel
2. maintain the current statuses and intervals for all channels
3. read the ADC channels periodically according to their statuses and intervals
4. transmit sets of readings made at about the same time (data packets) to the ground computer; these packets contain
    a. the number of channels in the packet
    b. the time the readings were made
    c. for each channel in the packet
        i. the channel identification
        ii. the reading for the channel
5. handle the message protocol for communications

The ground computer will:
1. transmit control packets to RDAS
2. receive data packets from RDAS
3. handle the message protocol for communications

The ADC has 64 channels, identified by the numbers 0 .. 63. It returns numbers in the range 0 .. 4095. RDAS will use 4096 to indicate a failed channel. A channel has

failed if the ADC has not responded within 0.001 seconds. RDAS takes no action when it detects a failed channel other than to transmit the failure value to the ground computer.

RDAS and the ground computer exchange messages. The message protocol is

stx message checksum etx

where checksum is calculated for the characters in message only. The format for a control packet (CP) message is

SCCIIIII

where

S indicates the desired channel status:
"O" indicates the channel should be open
"C" indicates the channel should be closed
CC is the channel identifier, "00" through "63"
IIIII is the interval for the channel in seconds, "00001" through "32400"

The format for a data packet (DP) message is

NNYYYYMMDDHHmmSSV

where

NN is the number of channels in the DP, "01" through "64"
YYYY is the year the readings in the DP where made ("1987")
MM is the month the readings were made, "01" through "12"
DD is the day of the month, "01" through "31"
HH is the hour, "00" through "23"
mm is the minute, "00" through "59"
SS is the seconds, "00" through "59"
V is the set of NN readings in the format CCRRRR, where
CC is the channel identifier, "00" through "63"
RRRR is the reading, "0000" through "4096"

After transmitting a message, both RDAS and the ground computer wait two seconds for an acknowledgment. If no acknowledgement is received, the message is retransmitted. If a NAK is received, the message is retransmitted. If an ACK is received, the next message is transmitted, if one exists.

Upon receiving a message, identified by the initial STX, both RDAS and the ground computer check for valid ETX and checksum, and then check the validity of the fields of the message. If all these checks are valid, an ACK is transmitted; otherwise, a NAK is transmitted.

Because RDAS may not be able to handle incoming characters while it is trying to acknowledge a CP or validate a CP, it must provide a buffer of at least 100 incoming characters. When the buffer is full, incoming characters are ignored.

Because the ground computer may not be accepting DP's because it is down or because communications may be interrupted, RDAS must provide a buffer of at least 50 DP's. When the buffer is full, a new DP will overwrite the oldest DP in the buffer. DP's will be stored internally in numeric format rather than ASCII format. RDAS will read channels as frequently as once every second if there are channels scheduled to be read.

EEG. The specification explicitly states that RDAS will interact with the ADC hardware. It clearly implies that RDAS will interact with communications hardware for receiving and transmitting messages. Less clear is that RDAS will need to access a clock to obtain time information. RDAS needs this information to supply in DP's, as well as to determine if it needs to make readings. RDAS will have to be able to perform time arithmetic to determine the next time to read an open channel.

RDAS will be built to use the Acme® Mark XIX communications hardware, but it should be easy to modify the system to use different communications hardware. This hardware is a "black box" which provides input and output functions. It makes sense to model this as a single entity.

Figure 4. shows the EEG for RDAS which contains this information. RDAS is active. It is also reusable, since anyone else with the same hardware and the same requirements could use RDAS as is. This is true of all systems, so the system under development is shown as reusable on the EEG for easy identification.

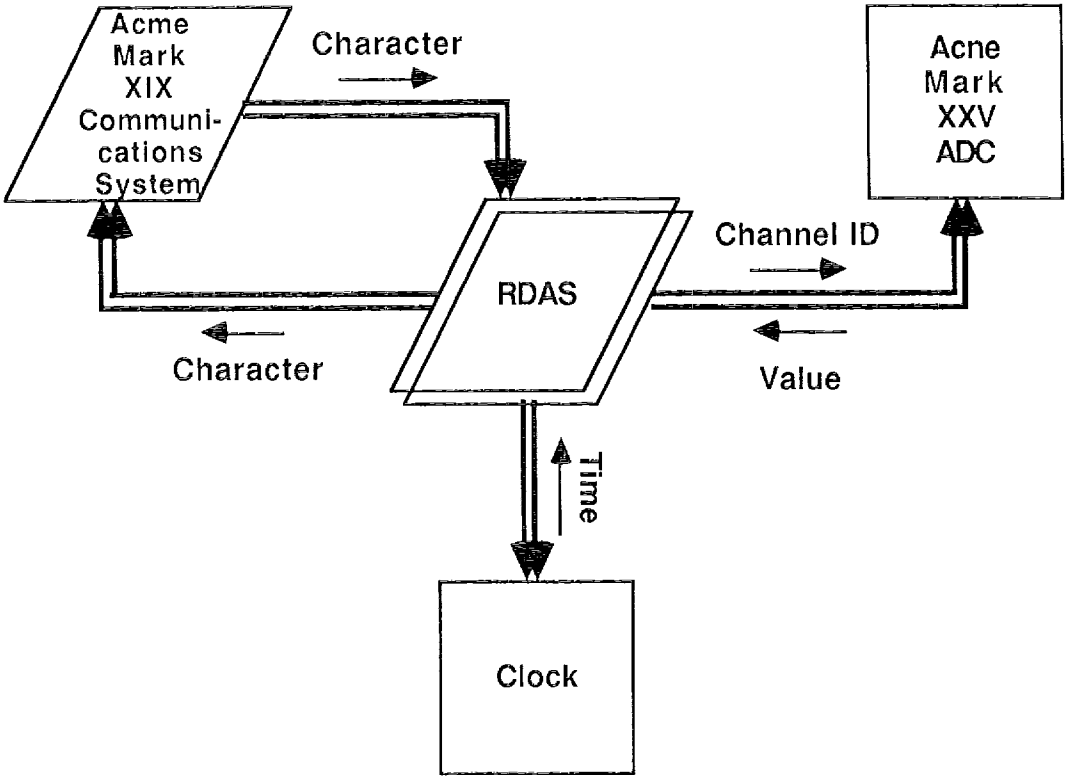

Figure 4.
RDAS External Entity Graph

Top-Level EIG. The next step is to determine the interface entities using an edges-in approach. Figure 5. shows the edge entities for RDAS. I was able to find an existing package to interface to the clock: package CALENDAR. Since this standard package should be familiar to everyone, I have not shown all of the interfaces into CALENDAR or the data flows. RDAS uses CALENDAR to obtain the time and to perform time arithmetic.

The communications interface is a generic instantiation and has a block on its internal interface for sending a character. These decisions have to be explained and recorded.

The entity is generic so it can call the system with an incoming character and still be application independent. The generic parameter is the procedure INCOMING, here instantiated as the buffer's PUT interface. The entity blocks calls to its outgoing interface when it is in the process of transmitting a character. Since I'm designing a generic entity, this information is formally recorded as

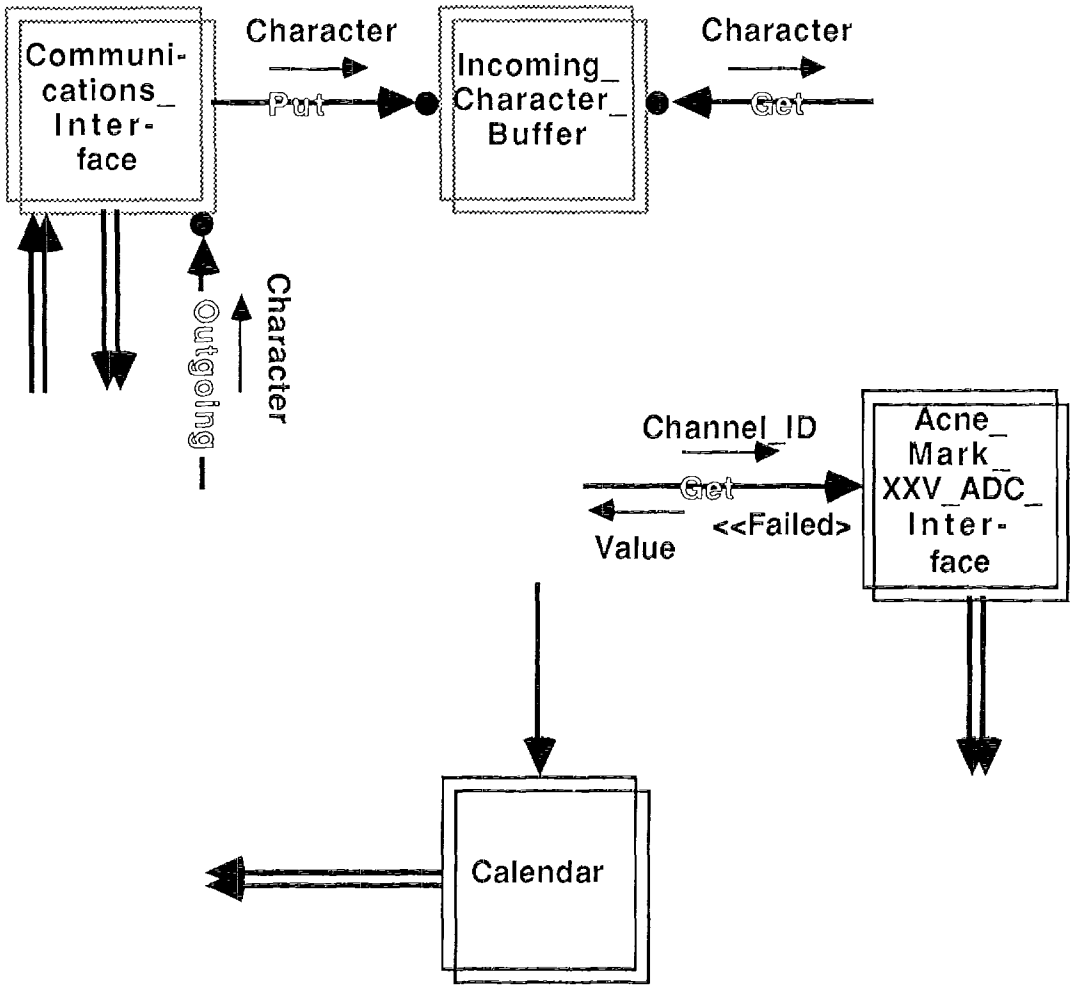

Figure 5.
RDAS Edge Entities

```
COMMUNICATIONS_INTERFACE:
        generic:    package
                        ACME_MARK_XIX_COMMUNICATIONS_INTERFACE
                    Reason:  Needs to call the system which uses it
                    Parameters:
                        with procedure INCOMING (CHAR : in CHARACTER);
        Interface   Attributes:
                    Blocks:
                        OUTGOING:  Cannot accept calls while transmitting
```

Such formal records should be made for every generic entity and for every entity with interface attributes or non-predeclared types.

The buffer for incoming characters is also shown. This is an instantiation of a standard blocking queue which allows concurrent access.

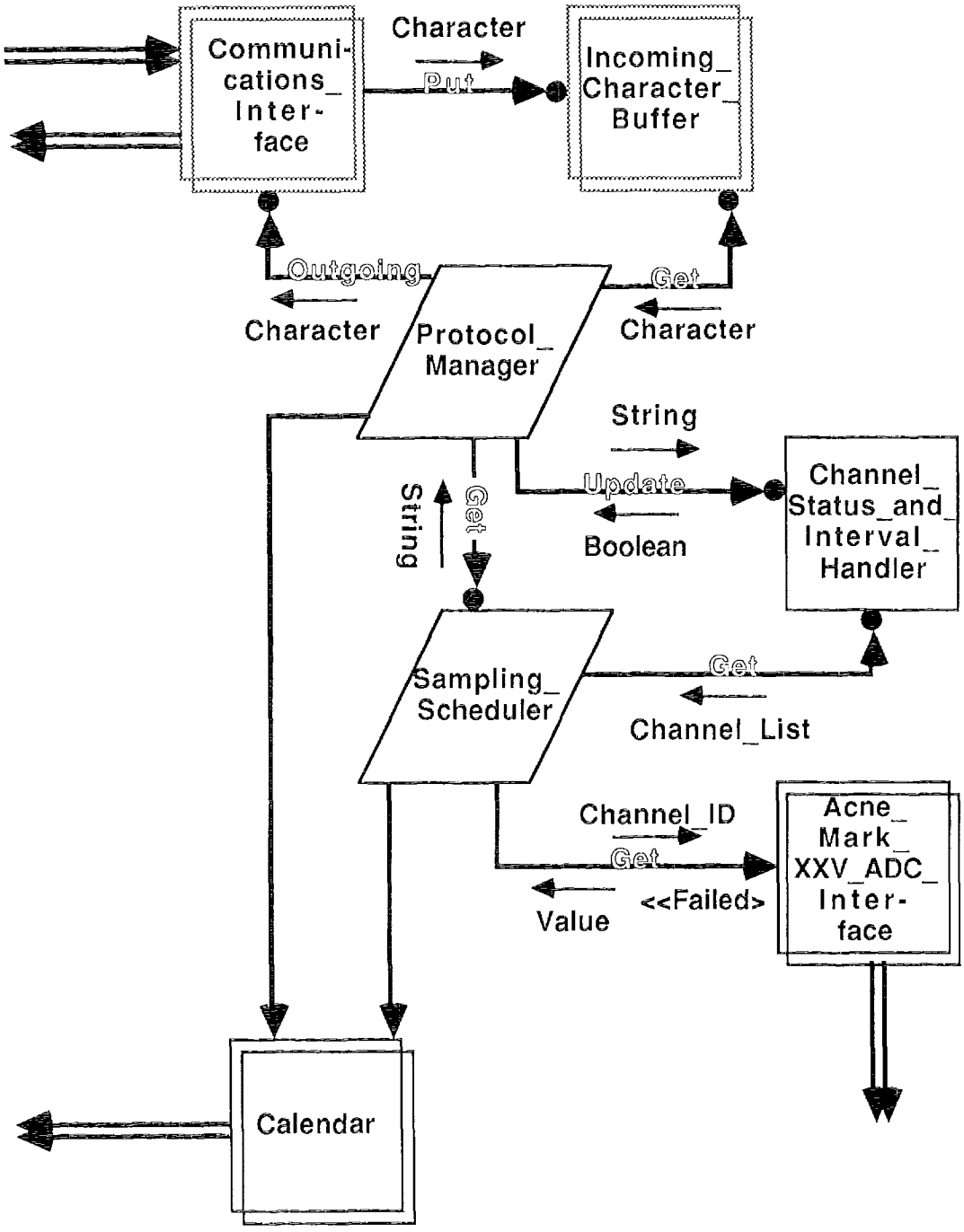

Figure 6.
RDAS Entity Interaction Graph

Now that the edge entities have been determined, the internal entities which connect them must be considered. From the specification, I can identify three main concepts which RDAS must handle and which are fairly independent. These concepts

are managing the message protocol, scheduling and reading the ADC, and maintaining the channels' statuses and intervals. A protocol manager can encapsulate all of the information about the protocol being used, allowing the protocol to be modified without impacting the rest of the system. A protocol manager will also handle all of the details of the protocol, such as sending and waiting for acknowledgements and calculating check sums. A sampling scheduler can schedule the readings of the ADC and buffer the readings in an internal format without any of the rest of the system knowing about the internal format, or even that the buffer exists. A channel status and interval handler can encapsulate the definition of what a valid CP message is, uncoupling this information from the protocol manager and allowing the format to be changed without affecting the rest of the system.

Figure 6. shows the RDAS EIG, the top-level EIG which decomposes the RDAS entity from the EEG, with these three entities added. Examination of a preliminary lower-level EIG for the protocol manager showed that it would need to access CALENDAR. The block on the sampling scheduler reflects the mutual exclusion of its internal buffer. The blocks on the Channel Status and Interval (CS&I) handler reflect that its interfaces are mutually exclusive in order to protect the integrity of its data: It cannot be updating its information at the same time that it is supplying that information.

It may not be apparent, but Figure 6. is Ada, in the sense that Ada code can be mechanically produced from the graph. The next step in developing RDAS is to produce and compile the Ada code which corresponds to Figure 6. This gives

```
generic
   with procedure INCOMING (CHAR : in CHARACTER);
package ACME_MARK_XIX_COMMUNICATIONS_INTERFACE is
   procedure OUTGOING (CHAR : in CHARACTER);
private -- ACME_MARK_XIX_COMMUNICATIONS_INTERFACE
   pragma INLINE (OUTGOING);
end ACME_MARK_XIX_COMMUNICATIONS_INTERFACE;

package ACNE_MARK_XXV_ADC_INTERFACE is
   subtype CHANNEL_ID is NATURAL range 0 ..   63;
   subtype VALUE      is NATURAL range 0 .. 4095;
   FAILED : exception;
   function GET (CHANNEL : CHANNEL_ID) return VALUE;
private -- ACNE_MARK_XXV_ADC_INTERFACE
   pragma INLINE (GET);
end ACNE_MARK_XXV_ADC_INTERFACE;

package CHANNEL_STATUS_AND_INTERVAL_HANDLER is
   subtype CHANNEL_ID is NATURAL range 0 .. 63;
   type CHANNEL_DATA is record
      OPEN     : BOOLEAN;
      INTERVAL : NATURAL;
   end record;
   type CHANNEL_LIST is array (CHANNEL_ID) of CHANNEL_DATA;
   function UPDATE (MESSAGE : STRING) return BOOLEAN;
   function GET return CHANNEL_LIST;
private -- CHANNEL_STATUS_AND_INTERVAL_HANDLER
   pragma INLINE (UPDATE);
   pragma INLINE (GET);
end CHANNEL_STATUS_AND_INTERVAL_HANDLER;
```

```
package PROTOCOL_MANAGER is
   -- null;
end PROTOCOL_MANAGER;

package SAMPLING_SCHEDULER is
   function GET return STRING;
private -- SAMPLING_SCHEDULER
   pragma INLINE (GET);
end SAMPLING_SCHEDULER;

with PROTOCOL_MANAGER;
procedure RDAS is
   -- null;
begin  -- RDAS
   null;
end RDAS;
```

CS&I Handler BTG. Of the three non-edge entities, only the CS&I handler is primitive. It spends most of its time waiting to be called, and then takes one of two different actions. This produces the BTG shown in Figure 7. The boxes represent the different behaviors taken by the entity. There are three: waiting to be called, updating, and providing the list. The arrows show the transitions from one behavior to another. There are two transitions out of waiting, so the arrows are labelled to show when each transition is taken. The other two behaviors only have one transition each, back to waiting. These arrows are unlabelled, which indicates that the transition is made as soon as the behavior has been completed. The unconnected arrow into waiting shows that it is the initial behavior. The label on this arrow shows that the CS&I handler initializes its list before entering the activity shown on the BTG. The initial state of the list is all channels closed.

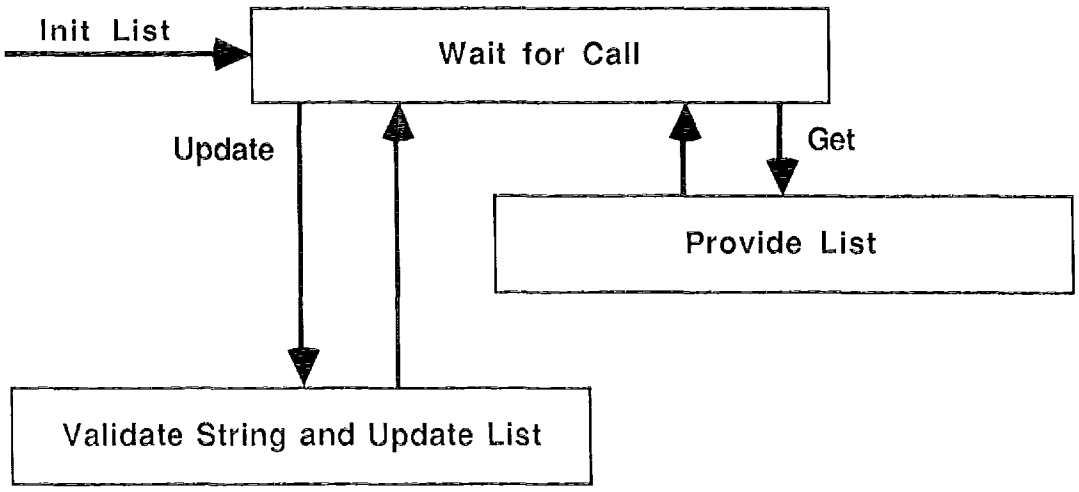

Figure 7.
CS&I Handler Behavior Transition Graph

The generation of Ada code from a BTG should be a simple, mechanical act. The BTG in Figure 7. produces

```
package body CHANNEL_STATUS_AND_INTERVAL_HANDLER is
   task HANDLER is
      entry UPDATE (MESSAGE : in STRING; VALID : out BOOLEAN);
      entry GET (LIST : out CHANNEL_LIST);
   end HANDLER;
   function UPDATE (MESSAGE : STRING) return BOOLEAN is
      VALID : BOOLEAN;
   begin -- UPDATE
      HANDLER.UPDATE (MESSAGE => MESSAGE, VALID => VALID);
      return VALID;
   end UPDATE;
   function GET return CHANNEL_LIST is
      LIST : CHANNEL_LIST;
   begin -- GET
      HANDLER.GET (LIST => LIST);
      return LIST;
   end GET;
   task body HANDLER is separate;
end CHANNEL_STATUS_AND_INTERVAL_HANDLER;

separate (CHANNEL_STATUS_AND_INTERVAL_HANDLER)
task body HANDLER is
   LIST : CHANNEL_LIST := (others => (OPEN => FALSE, INTERVAL => 0) );
   VALID    : BOOLEAN;
   CHANNEL  : CHANNEL_ID;
   INTERVAL : POSITIVE;
   STATUS   : CHARACTER;
   procedure CHECK (MESSAGE  : in  STRING;      VALID    : out BOOLEAN;
                    CHANNEL  : out CHANNEL_ID; INTERVAL : out POSITIVE)
   is separate;
begin -- HANDLER
   FOREVER : loop
      select
         accept GET (LIST : out CHANNEL_LIST) do
            LIST := HANDLER.LIST;
         end GET;
      or
         accept UPDATE (MESSAGE : in STRING; VALID : out BOOLEAN) do
            CHECK (MESSAGE => MESSAGE, VALID    => HANDLER.VALID,
                   CHANNEL => CHANNEL, INTERVAL => INTERVAL);
            VALID := HANDLER.VALID;
            if HANDLER.VALID then
               STATUS := MESSAGE (MESSAGE'FIRST);
            end if;
         end UPDATE;
         if VALID then
            LIST (CHANNEL).OPEN := STATUS = 'O';
            if LIST (CHANNEL).OPEN then
               LIST (CHANNEL).INTERVAL := INTERVAL;
            else
               LIST (CHANNEL).INTERVAL := 0;
            end if;
         end if;
      end select;
```

```
   end loop FOREVER;
end HANDLER;

separate (CHANNEL_STATUS_AND_INTERVAL_HANDLER.HANDLER)
procedure CHECK (MESSAGE : in  STRING;       VALID    : out BOOLEAN;
                 CHANNEL : out CHANNEL_ID; INTERVAL : out POSITIVE) is
   VALID_LOCAL : BOOLEAN;
   VALUE       : INTEGER;
   MESSAGE_LENGTH : constant :=      8;
   MAX_INTERVAL   : constant := 32_400;
begin -- CHECK
   VALID_LOCAL := MESSAGE'LENGTH = MESSAGE_LENGTH;
   if VALID_LOCAL then
      VALID_LOCAL := MESSAGE (MESSAGE'FIRST) = 'O' or
                     MESSAGE (MESSAGE'FIRST) = 'C';
      if VALID_LOCAL then
         VALUE :=
            INTEGER'VALUE (MESSAGE (MESSAGE'FIRST + 1 .. MESSAGE'FIRST + 2) );
         VALID_LOCAL := VALUE in CHANNEL_ID;
         if VALID_LOCAL then
            CHANNEL := VALUE;
            VALUE :=
               INTEGER'VALUE (MESSAGE (MESSAGE'FIRST + 3 .. MESSAGE'LAST) );
            VALID_LOCAL := VALUE in 1 .. MAX_INTERVAL;
            if VALID_LOCAL then
               INTERVAL := VALUE;
            end if;
         end if;
      end if;
   end if;
   VALID := VALID_LOCAL;
exception -- CHECK
when others =>
   VALID := FALSE;
end CHECK;
```

Sampling Scheduler EIG. The sampling scheduler has to check the statuses and intervals of all the channels once a second. If any channels are due to be read, it must read the channels and create an internal DP, which it stores in a buffer. The GET interface to the buffer must connect to the higher-level GET interface provided by the sampling scheduler and called by the protocol manager, and must provide the translation from an internal DP to a string. Figure 8. shows the EIG for the sampling scheduler.

The interested reader will want to produce the Ada code for Figure 8. and for the other EIG's and BTG's in this example.

Sampler BTG. Sampler is a cyclic activity. Every second it checks to see if any channels are due to be read. If so, it reads the channels, producing an internal DP which it stores in the buffer. It then updates its record of when channels should next be read, and repeats. The BTG in Figure 9. reflects this.

Although Figure 9. is simple, the Ada code it produces is somewhat more complex. This is due to sampler needing to handle time, both for when it does its sampling and for when channels next need to be read, and due to sampler needing to maintain a record of when channels next need to be read.

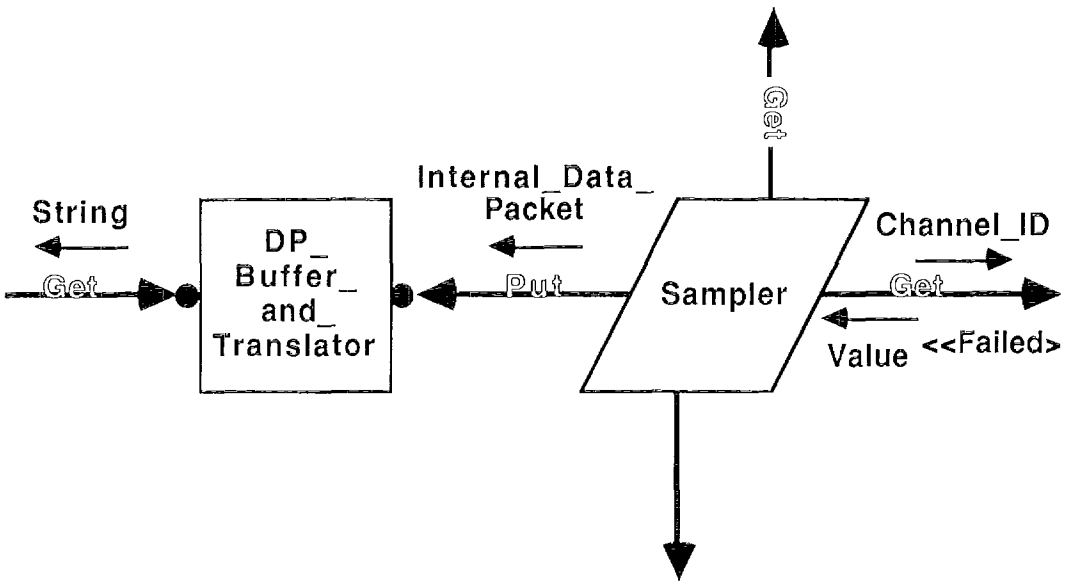

Figure 8.
Sampling Scheduler Entity Interaction Graph

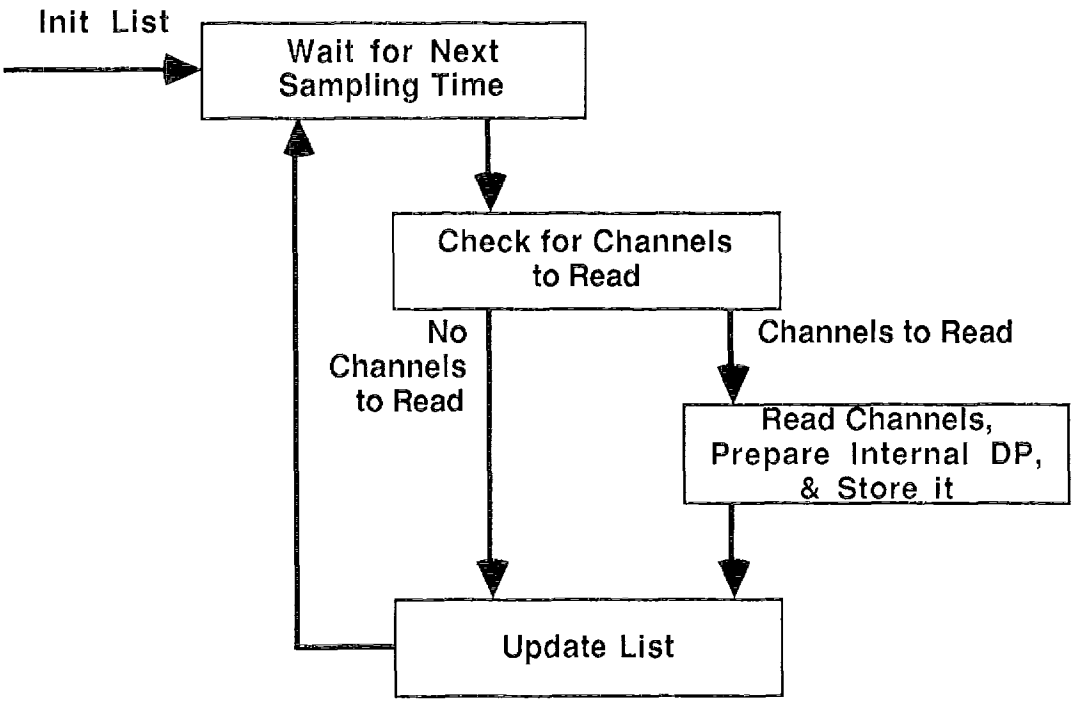

Figure 9.
Sampler Behavior Transition Graph

Protocol Manager EIG. The part of RDAS which I find most interesting is the protocol manager. This entity is inherently concurrent, since it must handle incoming

characters at the same time as it transmits outgoing characters. It must transmit both DP's and acknowledgements for CP's (CP ACK's). It must receive CP's, acknowledgements for DP's (DP ACK's), and garbage. It times out when waiting for a DP ACK. The DP ACK may arrive after it has timed out, in which case it should be ignored. The variety of its states and behaviors is richly varied and complex.

Figure 10. shows the EIG for the protocol manager. The message builder collects characters from the input into CP's. It also receives DP ACK's and hands them to the transmission scheduler via the latter's RECEIVE interface. The message builder validates the CP's it receives as regards the structure and check sum. If a CP is valid in these terms, the message builder passes it to the CS&I handler's UPDATE interface for content checking. If the CS&I handler returns TRUE, the message builder passes an ASCII ACK character to the transmission scheduler's TRANSMIT interface for transmission to the ground computer. If any of these checks fail, the message builder passes an ASCII NAK character to the transmission scheduler. If a character other than an ACK, NAK, or STX is received when the message builder is not constructing a CP, it passes a NAK to the transmission scheduler.

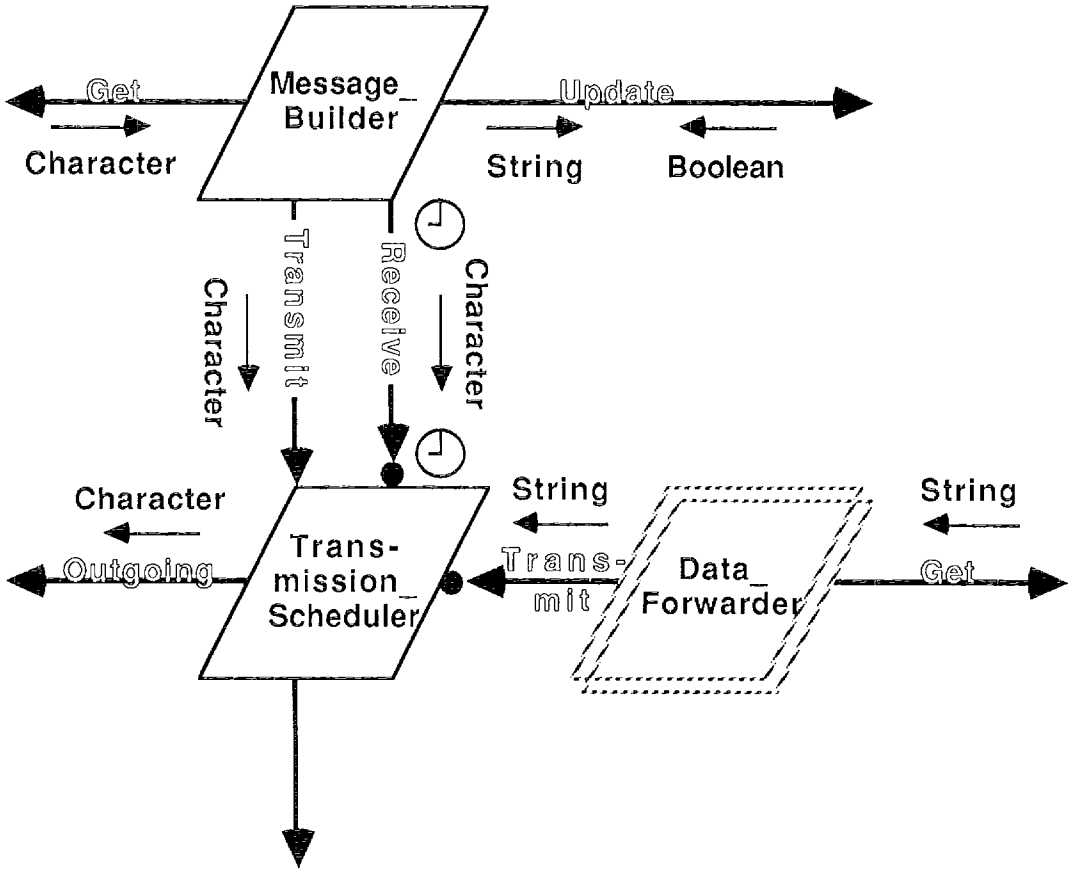

Figure 10.
Protocol Manager Entity Interaction Graph

The clock on the calling end of RECEIVE has an interesting impact on mechanically producing Ada code from Figure 10. In Ada, in order for a call to have a time out period, it must be a call to a task entry. The simplest and most straightforward way to mechanically produce Ada code from Figure 10. is for the transmission

scheduler to be implemented as a task which is visible to the message builder. Usually with MMAIM the developer has a choice, at least in principle, of whether an active entity will be implemented as a package or as a task. Although it is possible to achieve the same effect using a package, and a smart tool may be able to do so from Figure 10., the simplest translation represents the transmisssion scheduler as a task.

The asymmetry of the message builder and the transmission scheduler is interesting. The message builder is not called. It takes control and makes all the calls. The transmission scheduler, on the other hand, makes no calls until it is called. This is logical since the transmission scheduler needs to be able to accept either a CP ACK or a DP, whichever arrives first. The message builder only accepts characters from one source, so it can make a call to obtain them.

Message Builder BTG. The message builder gets an incoming character and decides what to to with it. If it's an ACK or a NAK, the message builder passes it to the transmission scheduler, assuming that it is a DP ACK. If the character's not an STX, the message builder tells the transmission scheduler to transmit a NAK. If it is an STX, the message builder collects the rest of the CP and checks it for valid structure and check sum. If these are invalid, the message builder tells the transmission scheduler to transmit a NAK. If they're valid, the message builder passes the message to the CS&I handler for content validation. If the CS&I handler says the message contents are invalid, the message builder tells the transmission scheduler to transmit a NAK, otherwise the message builder tells the transmission scheduler to transmit an ACK. This gives the BTG in Figure 11.

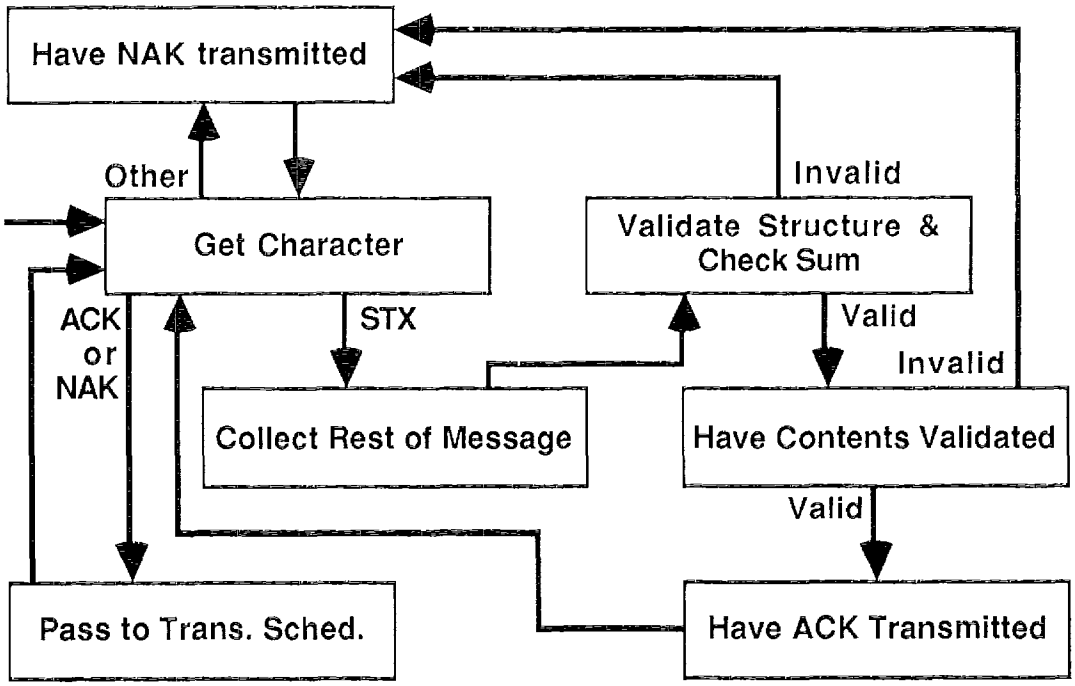

Figure 11.
Message Builder Behavior Transition Graph

Transmission Scheduler BTG. The transmission scheduler waits for a CP ACK or a DP to transmit. After transmitting a DP, it waits for a DP ACK to be received. While it's waiting for the DP ACK, it must also be able to transmit a CP ACK. If a NAK is received,

or if the timeout expires, the transmission scheduler retransmits the DP and waits for a DP ACK again. This produces the BTG in Figure 12.

Evaluation. That is RDAS. I have compiled this example for all of the non-hardware-dependent entities, and have written Ada pseudocode for the hardware-dependent entities. I will be happy to supply these to any reader who is interested in seeing MMAIM in more detail.

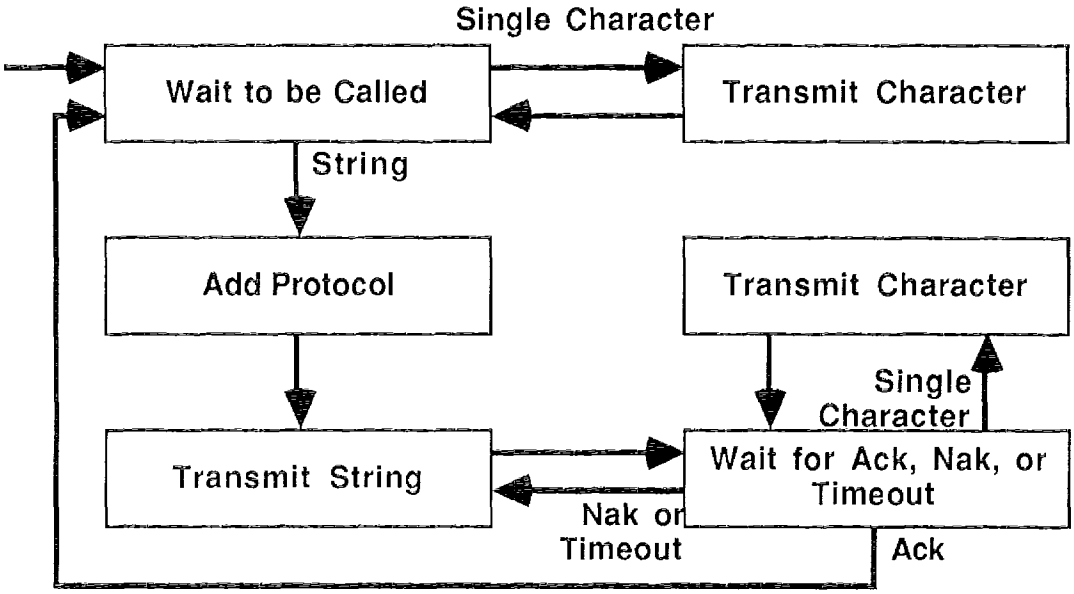

Figure 12.
Transmission Scheduler Behavior Transition Graph

It's useful to evaluate the quality of the solution MMAIM has produced. There are two main factors to the quality: Does it do the job, and is it easy to modify?

It's impossible to tell if the system will work without testing it, but it seems likely that RDAS will work. Whether it will meet the timing constraints given in the specification depends on the speed and number of processors on which it is run. Given the current trend in hardware prices, it is reasonable to expect that the implementation of real-time, embedded systems, such as RDAS, on multiprocessors will be the rule rather than the exception in the future.

The ease of modification is an interesting subject to consider. First, the likely modifications have to be identified, which may not be easy to do. I will mention three possible modifications to RDAS:

1. Changing the CP format to allow three characters for the channel identification, perhaps in anticipation of an ADC with more channels
2. Replacing the ADC hardware with a similar ADC with more channels and a greater range of returned values
3. Changing the DP format to YYYYMMDDHHmmSSCCVVVV, one reading per DP

Changing the CP format reveals a problem with the solution: The length of a CP message is declared in the CS&I handler's HANDLER task's CHECK procedure, but it is needed by the message builder for its "Collect Rest of Message" behavior. This shows that no method is perfect; all rely on intelligent decisions by the developer [9]. The

problem was not really due to MMAIM, but due to my decision to hide the length inside the CS&I handler. An improved system would make the length declared in the CS&I handler visible, so the message builder can use this declaration.

Changing the ADC turns out to be fairly simple. Suppose we decide to use the XIXCOR ADC. This ADC is very similar, in the way it is accessed, to the Acne ADC used in the solution, but it is more reliable, has more channels, has a greater range of returned values, and returns a reading sooner. Specifically, it has 100 channels identified by 1 .. 100, returns values in the range 0 .. 10,000, and returns a reading within 0.0001 seconds. The ground computer is going to be unchanged: It will still use channel identifiers of 0 .. 63 and expect values in 0 .. 4096. The values from the ADC should be converted to the values expected by the ground computer by the formula

(4095 * READING) / 10_000

The first change to make is to replace the Acne Mark XXV ADC entity on the RDAS EEG with a XIXCOR ADC entity which has an identical interface. Because of this change to the EEG we must replace the Acne Mark XXV ADC Interface entity on the RDAS EIG with a XIXCOR ADC Interface entity which appears identical to the Acne entity except for its name. The other change is to the Ada code for the sampler. The part of this Ada code which reads channels needs to perform the channel and value conversions.

When I made these changes, they were trivial. Ten statements were changed. The changes were isolated to one entity. None of the rest of the system needed to be changed: The sampler could be considered in isolation. This part of the system, at least, is of good quality.

The interested reader will want to work out what is involved in changing the DP format to one reading per DP. The size of a DP will decrease, but the number of DP's will increase, so the size of the DP buffer should increase so it can contain about the same amount of information. Note that this change will affect the sampler, the DP buffer, and the DP translator. The changes are still isolated to one entity: the sampling scheduler.

SUMMARY

The RDAS example demonstrates the use of MMAIM on a fairly large problem, and shows that the results of using MMAIM are robust and easily accommodate a major and likely change to the system.

REFERENCES

[1] Castor, V., and D. Preston, "Programmers Produce More with Ada," *Defense Electronics*, 1987 Jun.

[2] Nielsen, K., and K. Shumate, "Designing Large Real-Time Systems with Ada," *Communications of the ACM*, 1987 Aug.

[3] Carter, J., and C. McKeen, letter submitted to the "Forum" of the *Communications of the ACM*.

[4] Buhr, R., *System Design with Ada*, Prentice-Hall, 1984.

[5] Cherry, G., *PAMELA™ Designer's Handbook*, The Analytic Sciences Corp., 1986.

[6] Booch, G., *Software Engineering with Ada*, Benjamin/Cummings, 1983.

[7] Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Ada Lifecycle," in *Proceedings of Joint Ada Conference*, U. S. Army Communications--Electronics Command, 1987.

[8] Swartout, W., and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, 1982 Jul.

[9] Brooks, F., Jr., "No Silver Bullet," *Computer*, 1987 Apr.