The Design and Analysis of a Close-In Weapons System Simulator using HRT-HOOD*

Pete Cornwell, Alan Burns and Andy Wellings

Real-Time Systems Research Group,
Department of Computer Science
University of York

cornwell@minster.york.ac.uk
or
pcornwel@bournemouth.ac.uk

Abstract

This paper examines the architectural design and analysis of a naval close-in weapons system simulator. Using the TARDIS development framework we trace the emergence of a design solution using HRT-HOOD. We examine the formulation of a logical architecture, based on functional and non-functional *requirements* and demonstrate how this may be mapped to the constraints imposed by a particular target environment through a 'physical' process of timing and schedulability analysis.

*This work was funded by the Defence Research Agency (Maritime Division) as part of a "feasibility study of rigorous development methods for real-time software-intensive projects".

Glossary of Terms

Abbreviations

Block - *Blocking Time*
BRW - *Best Real-World*
CIW - *Close-In Weapon Event (from dedicated fire control radar)*
CIWS - *Close-In Weapons System*
DRA - *Defence Research Agency*
ES - *Electronic Surveillance*
ESA - *European Space Agency*
HRT-HOOD - *Hard Real-Time Hierarchical Object-Oriented Design*
LRR - *Long Range Radar*
PPI - *Plan Position Indicator*
WCET - *Worse Case Execution Time*

Time Units

sec - *Seconds*
ms -*Milliseconds*
μs - *Microseconds*

Unless explicitly stated all times are in *milliseconds*.

1.0 The Close-In Weapons System

The close-in weapons system case study (CIWS) is a simple naval command-control application managing point defence on board a British Type 42 destroyer. Designed to act as a last line of defence, the CIWS itself is a rotary machine cannon with a high rate of fire, that engages incoming missiles by simply attempting to shoot them down at short range. As part of a process of research undertaken on behalf of the Defence Research Agency (Maritime Division) [Glen 1992], the CIWS and its associated command control system were developed as a hard real-time case study utilising the HRT-HOOD design technique [Burns 1992] [Burns 1994a].

This report is divided into three complementary sections. The first is an introductory section that provides an overview of the case study requirements and constraints. The second examines the formulation of a requirements-based logical architecture for the CIWS. The third section demonstrates the translation from logical to physical design, where the logical requirements of the CIWS are resolved against the constraints imposed by an underlying 68020 target, through a process of timing and schedulability analysis.

1.1 System Requirements

The CIWS itself is supported by an infrastructure that collates information received by various sensors on the ship. These sensors are as follows:

- Long Range Radar (LRR) - A radar system thats gives bearing and distance of a given contact external to the ship.

- Electronic Surveillance (ES) - A system that attempts to invoke an IFF (*identification friend or foe*) transponder held aboard external 'friendly' vehicles. This sensor returns bearing.

- Own Ship Sensor (OSS) - The movements of the ship also have an effect on the bearing of various contacts external to the vessel. Therefore this is treated as a separate "sensor".

- Close-In Weapons Sensor (CIW) - A short range fire control radar mounted on the CIWS, returns bearing and distance.

From these four sources a *best real-world* or BRW picture is formed that *fuses* these contacts into distinct 'vehicles' external to the ship* . It is from this list of vehicles that targets are identifed and engaged by the CIWS.

The BRW picture is plotted to a display that contains the following elements:

- PPI (Plan Position Indicator) - A radar-like display with the ship plotted at the centre. Eight lines radiate out from the ship dividing the space around the ship into eight regular *sectors*. Two scaled range circles are drawn on the display

---

* The requirements actually state that only CIW events are received by the simulator. The HRT-HOOD graphical design decomposition (presented below) gives two versions: one with, and one without sensor fusion.

representing the *maximum attack range* (at 5000 metres) and the *maximum detection range* (at 25000 metres). It is here that the position of external vehicles are plotted and updated in real-time.

- Menu - A list of options which allow the user to interact with the system.

- Messaging Lines - Three text lines plotted at the top and bottom of the display. The first line (plotted at the top the display) records display configuration information such as orientation, magnification, mode of operation, and elapsed time since the simulation was started. The second and third lines (plotted at the bottom of the display) record the current user selected target, abort requests and attack messages from the CIWS.

Briefly, the display offers the following functionality to the user.

- Vehicle Selection - the user may click on any plotted vehicle on the PPI to select / deselect it as a target (except the ship itself!).

- Display Orientation - the PPI may be oriented as either NORTH_UP (with the top of the display pointing to magnetic north) or OWN_SHIP_UP (with the top of the display oriented in the direction of the ship's bow).

- Display Zoom / Unzoom - The PPI may be zoomed / unzoomed from 1.0 to 5.7 times magnification.

- Text Small / Large - Message text on the display may be plotted in a 7 or 25 point font.

- Icon Zoom / Unzoom - Vehicle icons on the PPI may be magnified / unmagnified to increase or decrease visibility.

- System Mode - This toggles the CIWS between MANUAL (where the user selects targets to engage) and AUTOMATIC (where the system selects targets to attack).

- Select / Deselect Sector (1-8) - This selects or deselects one of the eight sectors around the ship (the sector becomes highlighted in red). A selected sector is ineligable for target selection (in AUTOMATIC mode) by the CIWS.

- Abort - The user may use this option to abort an attack in progress.

- Quit - The user ends the simulation and dumps a timing log for later examination.

As the system mode function listed above implies, the CIWS has two modes of operation, MANUAL and AUTOMATIC respectively. In MANUAL mode the choice of target is determined by the user (by selecting the appropriate vehicle icon on the PPI). In AUTOMATIC mode the system hunts for targets itself, although the user is allowed to interrupt this process at any point by selecting a target him / herself. This will then be attacked in preference to any other target - but when destroyed or aborted the system will continue to hunt targets automatically.

For a vehicle plotted on the PPI to be considered a target by the CIWS it must fulfill the following criteria for *eligability*:

- Range - The vehicle must be at or less than 5000 metres from the ship.

- Speed - The vehicle must be travelling at or greater than 200 m/s.

- Allegiance - The vehicle must be registered as a 'Neutral' or 'Hostile' ** contact.

In addition, if the system is in AUTOMATIC mode *the system* may not select a target if it is within a selected sector (although the user may manually select targets in either MANUAL or AUTOMATIC modes). A vehicle that is selected by the user, but not considered a "target" by the criteria outlined above is never attacked by the system.

When a target is selected for attack in any mode of operation the following attack sequence is initiated:

1. A *request abort* message is sent to the display - the user is given *two seconds* to select the menu abort option.

2. If an abort is not received within two seconds the CIWS attacks the target - after a further period of *one second* the target vehicle is removed from the PPI and assumed destroyed. A *contact lost* message is sent to the display.

3. If the abort is recieved within two seconds, the attack does not take place. In addition the target becomes ineligable for future selection by *the system* in AUTOMATIC mode (although the user may manually re-select it in any mode).

1.2 Hardware

In order to provide a realistic platform for the CIWS study a three node distributed architecture is specified in the system requirements. This is composed of the following elements:

- A Sun 3/50 - based around a M68000 architecture

- A 16.2 Mhz M68020 board

- A Ferranti VARS-H graphics display

The Sun 3/50 was responsible for the timed dispatch of sensor events. Containing the complete scenario log it represented the devices used by the ship to detect external vehicle contacts. The Sun3 supports 4 MB of memory.

The M68020 board housed the CIWS system itself. The system received contacts dispatched from the sun 3/50 and built them into a list of vehicle contacts. Utilising this information the CIWS hunted for viable targets manually or automatically selected (dependent on the mode of operation), and drove the VARS display unit. The

** Commonly known as a "baddie".

M68020 board runs at 16.2 Mhz and supports 3 MB of memory with 2 memory wait states.

The VARS-H graphics display facilitated user interaction with the CIWS. Driven through the use of the Graphics Kernel System (GKS) the VARS display was utilised to plot the current vehicle contacts and allow the operator to perform a number of basic system operations through a menu interface. These operations range from manipulation of the display through zoom or re-orientation functions, to key modal and abort features. The software that drove this node (from the 68020) was derived from prototype code originally developed by the YARD company.

The hardware was linked in a Sun3-68020-VARS series using asynchronous interrupt driven serial communications. The overheads imposed by this arrangement were intended to represent the *combat systems highway* linking the disparate elements of a CIWS in a "real world" implementation. The respective speeds of these links are as follows:

1. Sun 3/50 to M68020 board - 2400 Baud.

2. M68020 board to VARS-H display - 19200 Baud.

1.3 Tools

The design, analysis and implementation of the CIWS were supported through the utilisation of a set of development tools, these are:

- The HRT-HOOD graphical notation and ODS editors.
- The Ada worse case execution time tool (WCET).
- Schedulability analysis tools reflecting current research at York.
- York/Ada ESA M68020 cross compiler.
- York/Ada Sun3 compiler.
- Scenario Preprocessor
- Sun3 Event Dispatcher*

The graphical and ODS editor is a prototype tool for the formulation of object-oriented designs in HRT-HOOD. Written in C under Xview, the tool supports the formulation of hierarchical architectures, allows the association of textual information with each object component and provides a persistent store (through the use of an underlying ONTOS database). Although currently incomplete the toolset was invaluable in the formulation (and verification) of the final CIWS design.

The worse case execution time (WCET) tool facilitates the timing analysis of object code produced by the York/Ada compiler. This process is accomplished in three stages [Forsyth 1992][Burns 1994b]:

---

* This tool appears as the EVENT_DISPATCHER object in the logical architecture.

1. The compiler includes Ada structural data from the source code in the symbolic debugging section of each object file

2. Texts segments from the object code file are read by a dissembler which converts each machine instruction into a convenient internal representation.

3. The dissembler output is then analysed to form basic blocks at the assembler level. Using the structural data included in (1), a worse case flow graph is produced. The respective times of each assembler block are determined by table lookup.

The schedulability tool performs deadline monotonic schedulability analysis, based on the work of Audsley et al at the University of York [Audsley 1993a][Audsley 1993b]. The tool performs the test based on the real-time characteristics of application tasks and the overheads imposed by the scheduler and Ada runtime system.

Two versions of the York/Ada compiler were utilised for the implementation of the CIWS. The Sun3 compiler was used in the development of the timed event dispatcher, hosted on the Sun3 node, and is a standard Ada 83 compiler [Firth 1988]. The ESA cross compiler for the 68020 was specifically developed to support hard real-time development and therefore included many non-standard features, many of which will duly appear in the forthcoming real-time annex for Ada 95 [Burns 1993b][Firth 1992]. The modifications made to the current Ada standard that directly effected the CIWS design and implementation are summarised in section 2.6 on *terminal ODS specification*.

Contact information sent from the Sun3 to the 68020 node was read from a scenario file, that listed the type of contact and the time at which it was to be dispatched. In order to convert this text-based file into a suitable Ada-based representation we employed an independently developed scenario preprocessor tool. This tool converted the file into a set of CIW sensor contacts, by "fusing" LRR and ES event lists into a single data structure (in essence producing a "best real-world" picture of contacts around the ship). These CIW events were written into a sequential data file for extraction by the event dispatcher tool.

The event dispatcher tool faciliated the timed dispatch of CIW events from the Sun3 to the 68020 node. The tool was composed of two components:

- Setup - this was a simple C program that configured the serial port on the Sun3 as the standard output stream, and set up the line speed (at 2400 baud).

- Dispatcher - this was an Ada program that read the preprocessed scenario file, loaded it into an internal event list and converted it into a set of simple character-based packets for transmission to the 68020 node. As each CIW event was tagged with a dispatch time (a monotonic offset), the tool was able to send events to the 68020 node at times consistent with their listing in the scenario file.

2.0 The Logical Architecture

This section outlines the practical design decisions and process that culminated in the development of the CIWS logical architecture. Particular emphasis will be placed on the hierarchical "top-down" development process central to the HRT-HOOD design

philosophy and the gradual evolution of the architecture in response to underlying constraints imposed by the target environment, and a greater understanding of the application domain.

2.1 The Root Level

The top level of any HRT-HOOD design tree is known as the *root object*. This object represents the system to design at the highest level of abstraction, and may be subsequently refined through a recursive process of structural decomposition into a detailed set of encapsulated child objects, which may themselves be decomposed and so on.

The root level of the CIWS is an ACTIVE object simply called 'SYSTEM' (fig 1). SYSTEM does not offer 'services' to external objects within the design space (i.e. constant, exception, operation and type definitions) and therefore does not support a PROVIDED_INTERFACE.

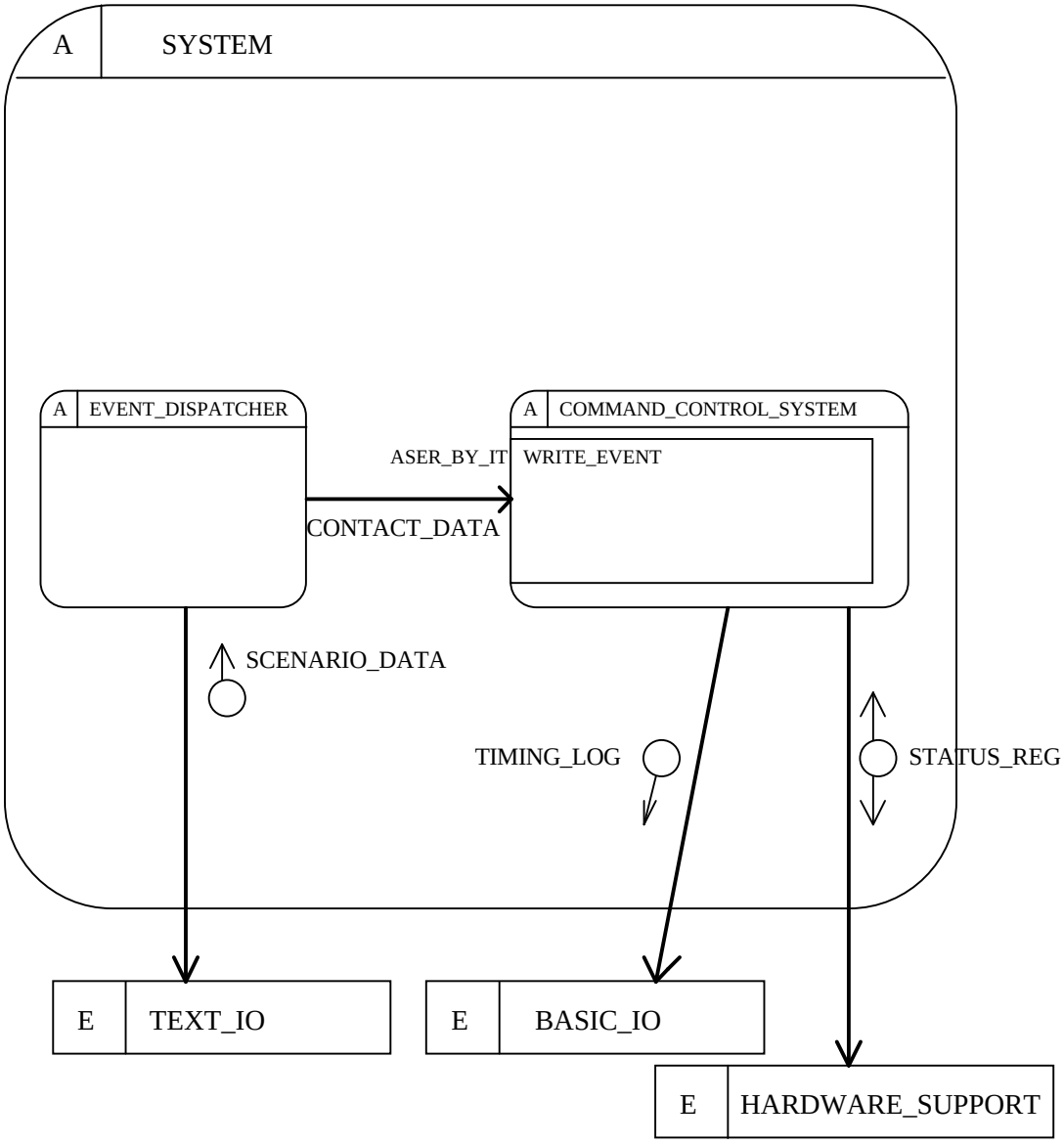

fig (1) The Root Level SYSTEM object

The distributed configuration of the target hardware, seemed to indicate a 'natural' decomposition of the SYSTEM object into two child objects. These objects are represented in the design as:

- EVENT_DISPATCHER (ACTIVE)

- COMMAND_CONTROL_SYSTEM (ACTIVE)

This decomposition was made as it was felt that each child object had a distinct area of responsibility. The EVENT_DISPATCHER object, residing on the Sun3 node, would read contact events from a scenario file and then dispatch them to the COMMAND_CONTROL_SYSTEM object residing on the 68020 board. The COMMAND_CONTROL_SYSTEM object would implement the 'core' of the system, driving the display, managing contact information and attacking valid targets.

This relationship is formalised by a *use* relationship between the two objects. EVENT_DISPATCHER uses the single Write_Event operation declared in the PROVIDED_INTERFACE of COMMAND_CONTROL_SYSTEM. There is also an *in* data flow of 'contact_data' between these objects.

Three environment objects also appear at the root level of the design, these are:

- TEXT_IO (PASSIVE) - this environment object represents the standard Ada package for the management of I/O on the Sun3 node, and is used by the EVENT_DISPATCHER object to convert, read and dispatch contact data from the scenario file.

- BASIC_IO (PASSIVE) - this environment object represents the standard I/O package for serial line communication between the 68020 and the Sun3 or VARS nodes and is used by the COMMAND_CONTROL_SYSTEM to send control characters to the VARS-H display or dump a timing log at the end of the simulation to the Sun3.

- HARDWARE_SUPPORT (PASSIVE) - this environment object provides facilities for the enabling and disabling of interrupts generated by MC68681 serial port controller on the 68020 board.

2.2 EVENT_DISPATCHER

The EVENT_DISPATCHER object is a high level representation of the dispatcher tool, that sent CIW_EVENTS from the Sun3 to 68020 node.

2.3 COMMAND_CONTROL_SYSTEM

The COMMAND_CONTROL_SYSTEM object encompasses the bulk of the system specified in the original CIWS requirements. As previously mentioned its responsibilities encompass the management of the VARS-H display, the organisation and processing of contact information, and the close-in weapons system itself which may attack targets in either automatic or manual modes of operation.

This section of the design underwent significant refinement between the initial design [Cornwell 1993] and the present logical architecture. The reader is invited to compare and contrast the original decomposition (fig 2) with the finalised design (fig 3).

The original design decomposed COMMAND_CONTROL_SYSTEM into three child objects, these were:

- SENSORS (ACTIVE) - a repository for incoming contact events dispatched from the Sun3.

- USER_INTERFACE (ACTIVE) - an object that managed the users interaction with the VARS-H display.

- CIWS (SPORADIC) - the close-in weapons system itself.

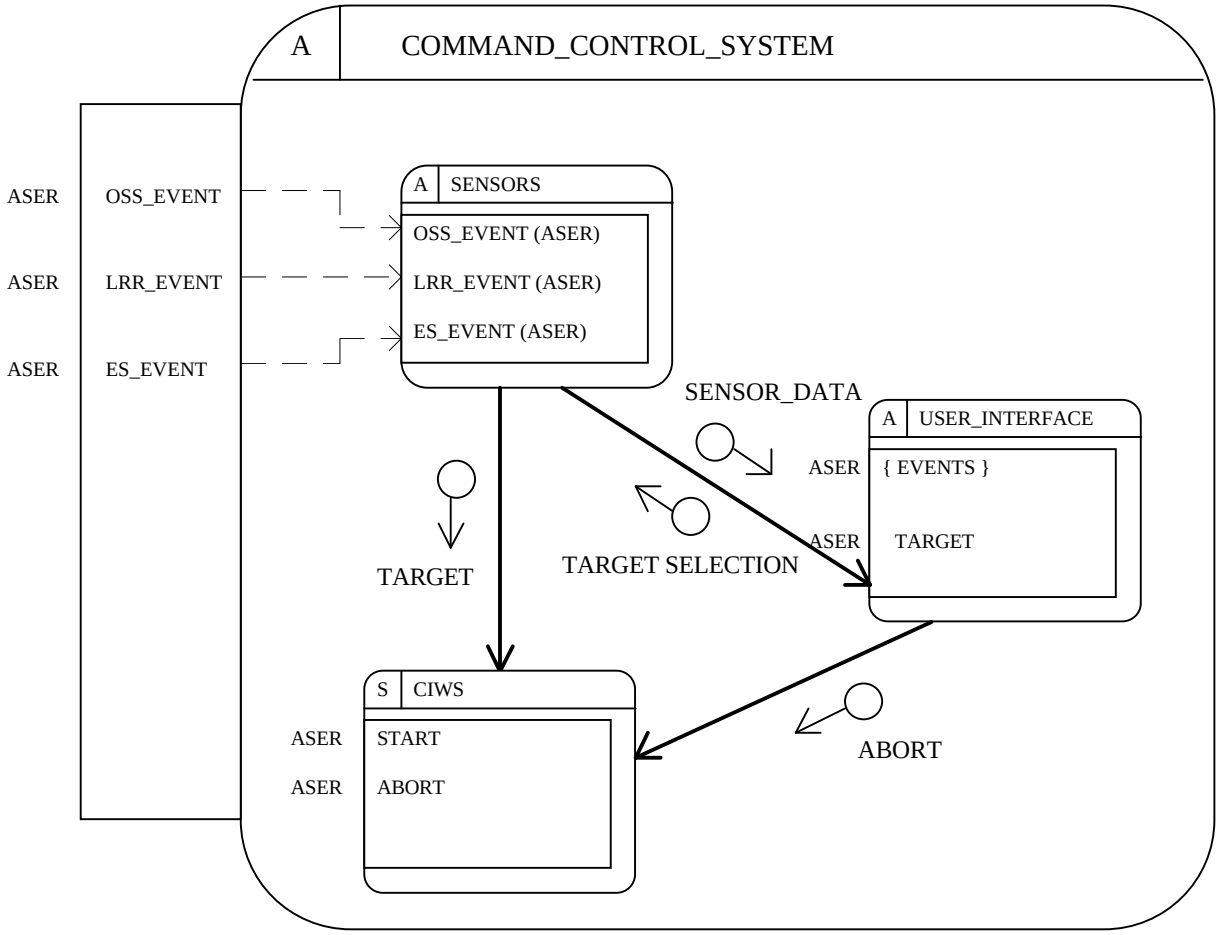

fig (2) The Original COMMAND_CONTROL_SYSTEM decomposition

In many respects this was a satisfactory decomposition, and many elements survived to appear in the finalised design. Each child object had a reasonably well defined area of responsibility. The SENSORS object implemented the original interface of its parent, providing a repository for incoming contact information which could be *used* by USER_INTERFACE. Furthermore this object would scan its internal contact lists, retrieve appropriate targets and dispatch them to the SPORADIC CIWS object which would then engage them, subject to the receipt of an abort request.

The USER_INTERFACE object managed the display, retrieving contact information from SENSORS and informing that object of changes to the target acquistion process (i.e. manual or automatic) or manual selection of a given contact. This object also informed the CIWS object of a user abort.

However, it soon became clear that this original design was not wholly satisfactory for four reasons:

1. The SENSORS object did not represent a particularly cohesive abstraction. It was simultaneously a repository for incoming contact data, the Sun3-68020 communications interface and the determinant for eligable targets dispatched to the CIWS.

2. The dialogue between USER_INTERFACE and CIWS was strictly one way (i.e. through the dispatch of an abort request), the user would need to be informed of the progress of the CIWS through text written to the display, such as requests for abort, engaging target and contact lost messages. All of which necessitated a rapid two-way dialogue not reflected in the original design.

3. Although a simple polling mechanism had sufficed to handle control characters sent from the VARS to the 68020 in the original YARD CIWS prototype, a full implementation with a greater task load would be unable to poll the appropriate port fast enough to prevent character loss through subsequent overwriting (the MC68681 serial port controller only has a 3 character buffer). Instead it was decided that *all* serial port communications should be interrupt-driven to prevent data loss. However the the MC68681 only allows the association of a single interrupt vector for *both* ports, which would necessitate handling input from the VARS and Sun3 at the same point in the design (i.e through a single interrupt handler). This was not reflected in the original design decomposition.

4. As a scenario preprocessor tool was utilised to "pre-fuse" incoming contacts, only CIW events would need to be received by the system. The LRR_EVENT, OSS_EVENT and ES_EVENT operations provided by SENSORS therefore became redundant.

To resolve these issues the COMMAND_CONTROL_SYSTEM object was re-configured at the child level. Although many of the original design features were retained, the role of each object was significantly revised to produce a more cohesive (and hence comprehendable) design decomposition. This resulted in the following child objects:

- COMMS_RESOURCE (ACTIVE) - This object handles serial communication with both the Sun3 and VARS display. Contact data is placed in the DATA_STORE object for eventual retrieval and use by other objects in the system. GKS commands from the VARS-H display are stored for retrieval by the USER_INTERFACE object.

- DATA_STORE (PROTECTED) - This object acts as a central repository for contact and system mode information and is used by both the CIWS and USER_INTERFACE objects.

- USER_INTERFACE (ACTIVE) - This object drives and manages user interaction with the VARS display. Contacts are retrieved from the DATA_STORE object and plotted to the display. A two way dialogue is also faciliated with the CIWS via the PROTECTED DIALOGUE_CONTROLLER object.

- CIWS (ACTIVE) - This object scans the DATA_STORE object for eligable targets. If an eligable target is retrieved the system will begin to attack. The CIWS will block on either Abort_Fore_Check or Abort_Aft_Check in DIALOGUE_CONTROLLER for two seconds awaiting an abort request from the USER_INTERFACE. If nothing is received during that period the CIWS will engage the retrieved target. Messages indicating the current status of the CIWS are also asynchronously refered to the USER_INTERFACE via the DIALOGUE_CONTROLLER.

- DIALOGUE_CONTROLLER (ACTIVE) - This object controls the dialogue between the CIWS and USER_INTERFACE objects. This dialogue encompasses both the user abort and messages sent from the CIWS to the display.

Much of the original functionality attributed to the SENSORS object was disseminated between the CIWS , DATA_STORE and COMMS_RESOURCE objects that appear in the final logical architecture.

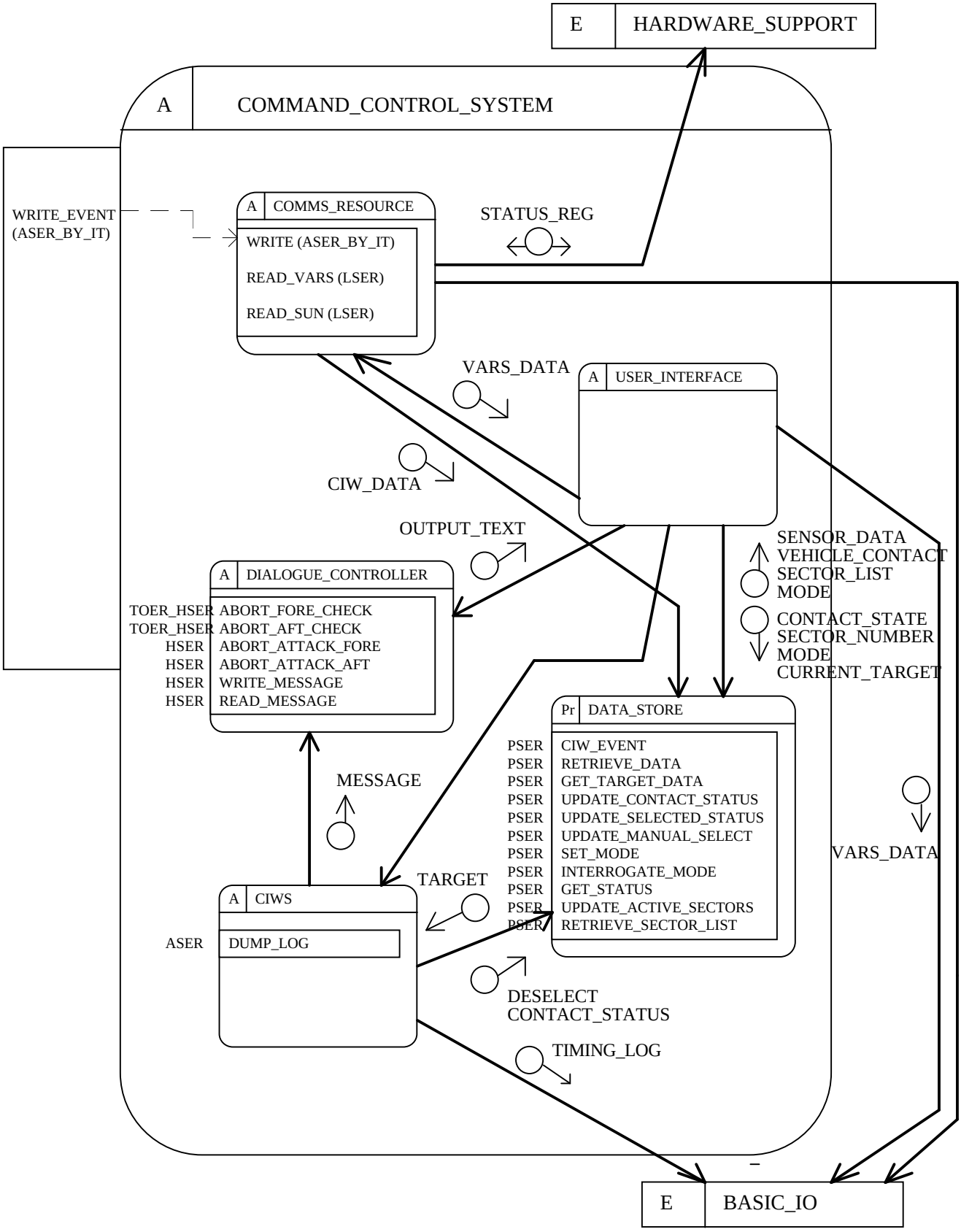

fig (3) The COMMAND_CONTROL_SYSTEM object

2.4 COMMS_RESOURCE

This object encapsulates the terminal level children that capture, decode and store input from the Sun3 and VARS-H system nodes (fig 4). These child objects are:

- COMMS_MANAGER (PROTECTED) - An object that handles interrupts from both serial ports on the 68020 board (linked to the Sun3 and VARS-H nodes). It also encapsulates control characters sent from the VARS box to the 68020 board (for asynchronous consumption by the USER_INTERFACE object) and contact data sent from the EVENT_DISPATCHER object (for asynchronous consumption by the COMMS_CONSTRUCTOR object).

- COMMS_CONSTRUCTOR (CYCLIC) - An object that extracts contact data packets from the COMMS_MANAGER. This data is unpacked by the PASSIVE DATA_CONSTRUCTOR object, returned, and then forwarded to the DATA_STORE object, which appears as an uncle object on this diagram.

- DATA_TRANSLATOR (PASSIVE) - This object unpacks and then returns contact information sent to it by COMMS_CONSTRUCTOR.

This simple and satisfactory design solution was fairly easily arrived at. Data is received and stored by the COMMS_MANAGER object which maintains two lists of information, one pertaining to information recieved from the VARS-H, the other storing contact information recieved from the Sun3 EVENT_DISPATCHER. As the VARS commands are only used by the USER_INTERFACE object this information may be retrieved by that object calling the read_vars ASER operation in COMMS_RESOURCE.

In contrast contact data must be made visible to both the USER_INTERFACE and the CIWS objects which necessitates its avaliability in the central repository represented by DATA_STORE. The CYCLIC COMMS_CONVERTER object facilitates this by extracting a packet of information pertinent to each distinct sensor contact, having this converted, then forwarding it to DATA_STORE for system-wide consumption.

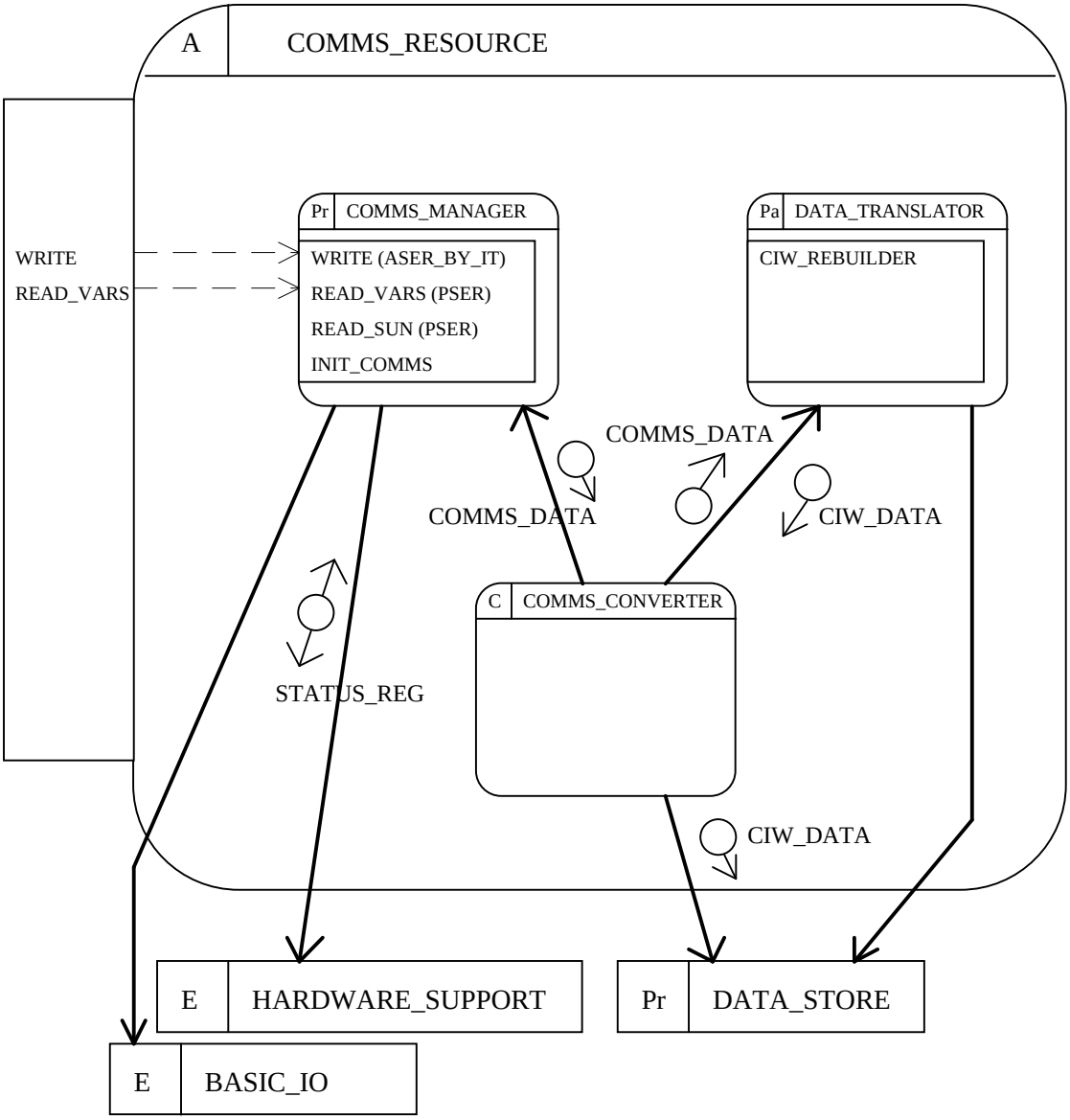

fig (4) The COMMS_RESOURCE object

2.5 DATA_STORE

The DATA_STORE object is the central repository for contact data dispatched from the scenario file on the Sun3 node. During the logical design phase this object evolved from the less cohesive SENSORS abstraction into its final form. In order to trace this evolution and examine the underlying design decisions that formulated this core part of the system we present three designs for this object, these are:

1. The original SENSORS object.

2. The final DATA_STORE object. This is the object that appears in the final CIWS implementation, but assumes the existence of a *scenario pre-processor* on the Sun3 node.

3. An 'ideal' DATA_STORE object. This object receives contact data from LRR , ES and OSS sources and fuses them to produce a 'best real world' picture of the environment surrounding the ship.

As aforementioned the SENSORS object had four areas of responsibility:

1. Receiving contact data from the Sun3 node.

2. A central repository for contact information.

3. A basis for the fusion of contact data.

4. A determinant and dispatcher of targets to the CIWS (which was a SPORADIC in the original design - see above).

As well as being less than fully cohesive, the object also failed to encompass the interrupt vector constraint identifed earlier, as it only made provision for the receipt and storage of information from the Sun3 node only. The SENSORS object was decomposed into the following children (fig 5):

- OSS_CONTROLLER (SPORADIC) - This object would handle interrupts pertaining to the arrival of OSS (Own Ship) events.

- LRR_CONTROLLER (SPORADIC) - This object would handle interrupts pertaining to the arrival of LRR (Long Range Radar) events.

- ES_CONTROLLER (SPORADIC) - This object would handle interrupts pertaining to the arrival of ES (Electronic Surveillance) events.

- SENSOR_BASE (ACTIVE) - This object would both act as a repository for incoming contact data, produce and dispatch target data to the CIWS (data which is referred to as a CIW event).

The SPORADIC handlers that handled and dispatched contact events to the SENSOR_BASE object became obsolete when the underlying interrupt constraints were factored into the design. It was also felt that the overhead imposed by three distinct threads of control on the implementation would be prohibitive. Instead this element of the design became part of the COMMS_RESOURCE object and its children (see above), where the single interrupt channel was serviced by an unthreaded PROTECTED interrupt handler (COMMS_MANAGER) and unpacked by the CYCLIC child object COMMS_CONVERTER. All in all a much more efficent arrangement in terms of overall performance.

The SENSOR_BASE child object was the least cohesive aspect of the parent, as it managed sensor fusion, target identification and dispatch. In the final , stable version of the system the logical architecture was reconfigured to place the responsibility for target selection and retrieval within the CIWS object itself which appeared to be a more comprehendable and cohesive design solution.

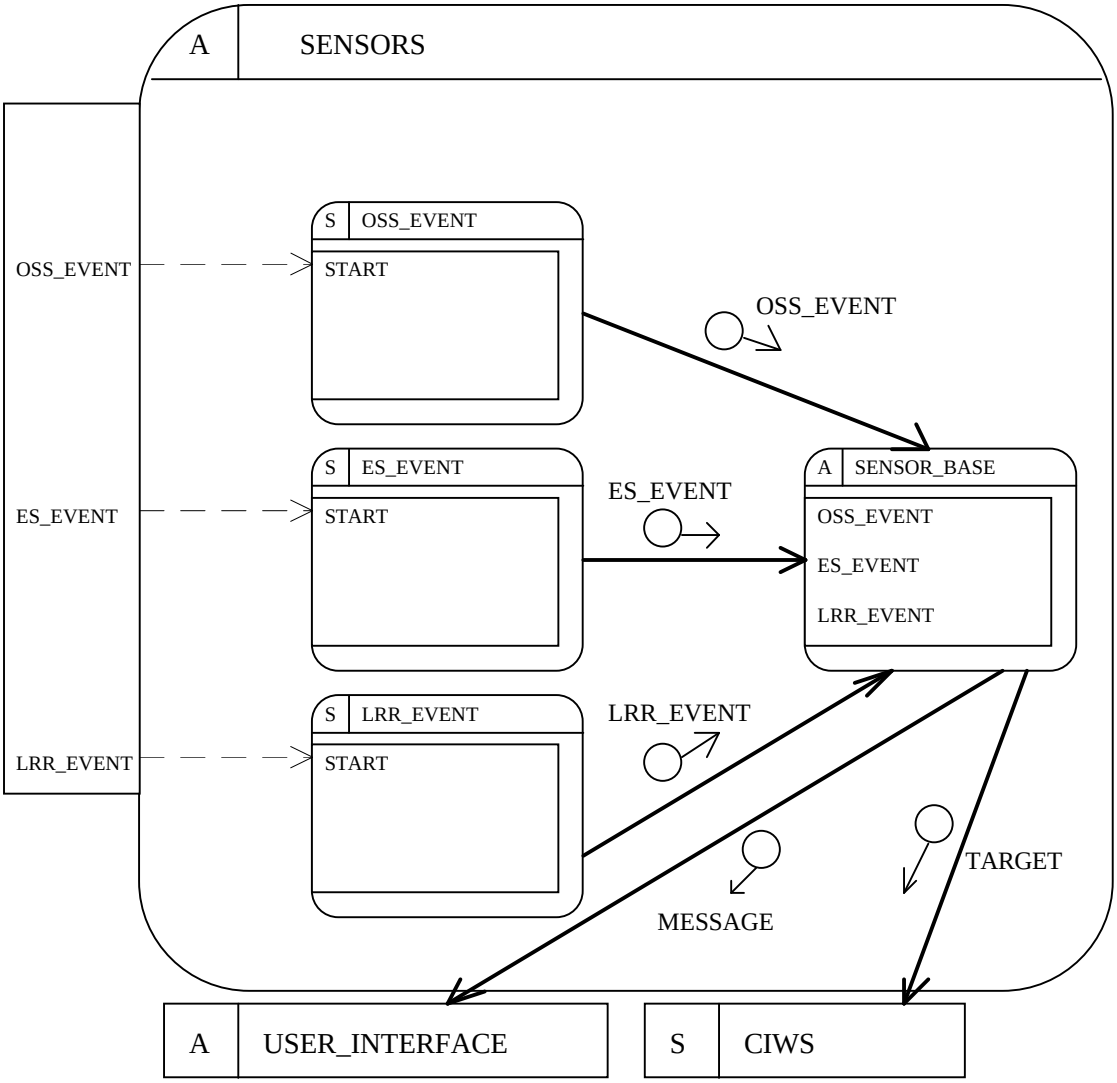

fig (5) The original SENSORS object

This overall change in emphasis resulted in the formulation of the DATA_STORE object (fig 6).

The two versions of the design presented below are both equally valid solutions - one represents the *actual* solution, where the scenario events were pre-processed to extract valid targets for the close-in weapons system to engage. The other is an *ideal* solution where LRR and ES contacts are fused on 'the fly' by the system to produce eligable targets for the system to attack. The ideal solution decomposes into two child objects:

- CONTACT_STORE (PROTECTED) - This object is a repository for OSS , LRR, ES and CIW events. It also encapsulates data pertinent to sector management and the current mode of operation (MANUAL or AUTOMATIC modes).

- CIW_CONSTRUCTOR (CYCLIC) - This object extracts arriving LRR and ES events and fuses them to produce valid CIW contacts, placing these back into CONTACT_STORE for consumption by the CIWS object.

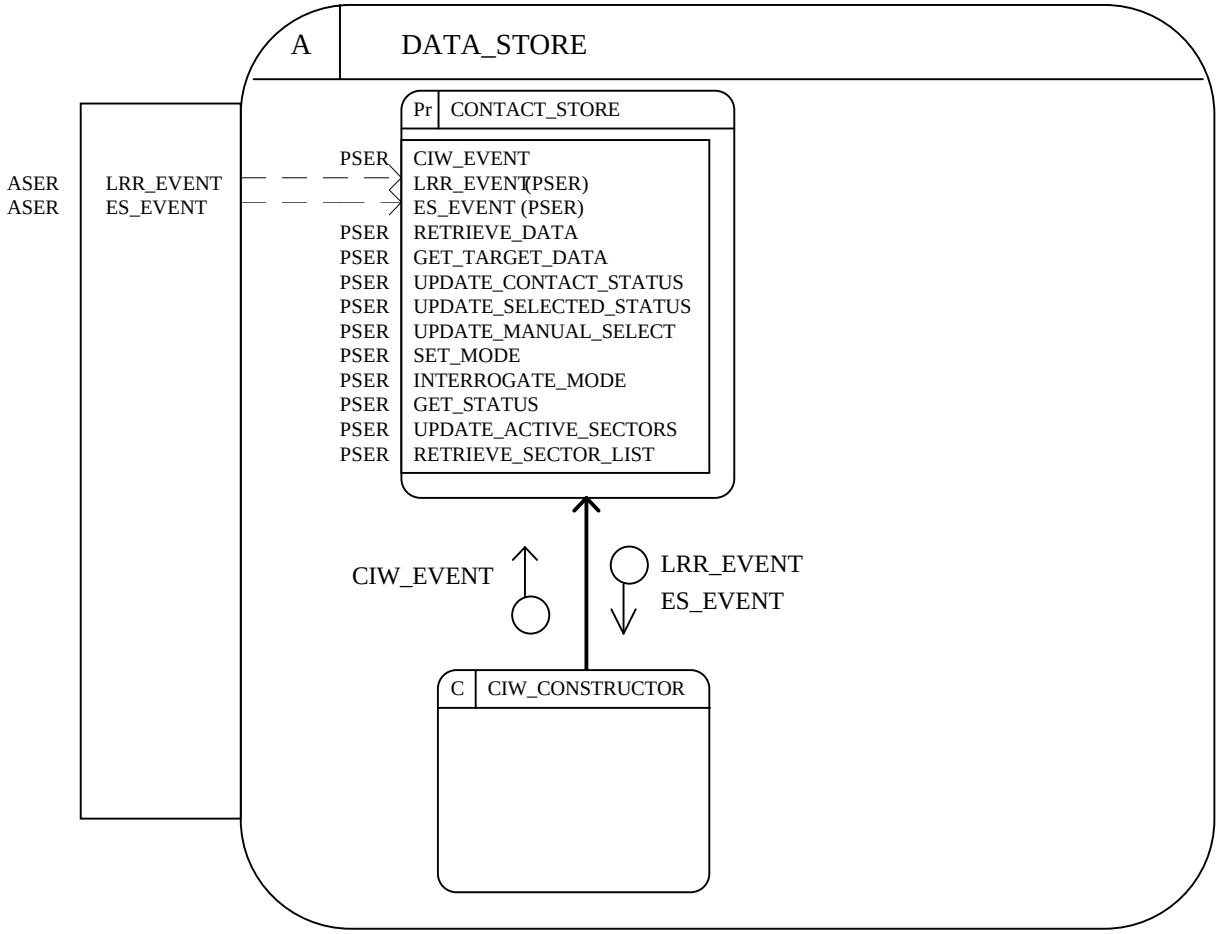

fig (6) The "ideal" design decomposition (with sensor fusion)

The actual solution removes the CIW_CONSTRUCTOR object (as the data arrives *pre-fused*). In this case decomposition of the DATA_STORE object becomes unnecessary (as it would only have a single child object). Instead the actual logical design considers DATA_STORE to be a terminal object, and promotes it from an ACTIVE to a PROTECTED type. In essence CONTACT_STORE in the ideal design becomes DATA_STORE at a higher level of abstraction.

2.6 USER_INTERFACE

This object manages the VARS-H display and handles interaction between the user and the system. Its responsibilities include:

- PPI Orientation (either NORTH_UP or OWN_SHIP_UP)
- Display Magnification (from 1.0 to 5.7 times magnification)
- System Mode (either MANUAL or AUTOMATIC operation modes)
- Manual Target Selection (Selecting a contact with the pointer device).
- Abort (Aborting an attack on an eligable target).

- Sector Management (making one or more octal segments around the ship ineligable for AUTOMATIC target selection)

- Messaging (Informing the user of the state of the CIWS through text messages)

As this section of the system was already supplied for us (through pre-existing Ada code written by YARD), the code was manually reverse-engineered and adapted for our purposes. This resulted in the following 'design decomposition' (fig 7):

- VARS_DRIVER (CYCLIC) - handles incoming events from the VARS (by reading incoming data from COMMS_RESOURCE), extracts and plots contact data from the DATA_STORE repository and manages the display functionality.

- VARS_IO (PROTECTED) - this object used by VARS_DRIVER retrieves data from the VARS (via COMMS_RESOURCE) and outputs directly to the display through BASIC_IO.

- GKS (PROTECTED) - this object encapsulates the primitives and data used to draw to the display.

Our modifications to this design have primarily involved linking the USER_INTERFACE object with both the DATA_STORE (to update the display with current sensor contacts) and the CIWS (to abort an attack or inform the user of the state of the CIWS).

The VARS_DRIVER therefore interacts with the external system in four ways:

1. Incoming data from the VARS-H display is retrieved from the PROTECTED object COMMS_RESOURCE (through VARS_IO).

2. Contact and mode is retrieved from the central DATA_STORE object.

3. Messages from the CIWS object are read via the PROTECTED object DIALOGUE_CONTROLLER.

4. Abort requests are sent to the CIWS via the PROTECTED object DIALOGUE_CONTROLLER.

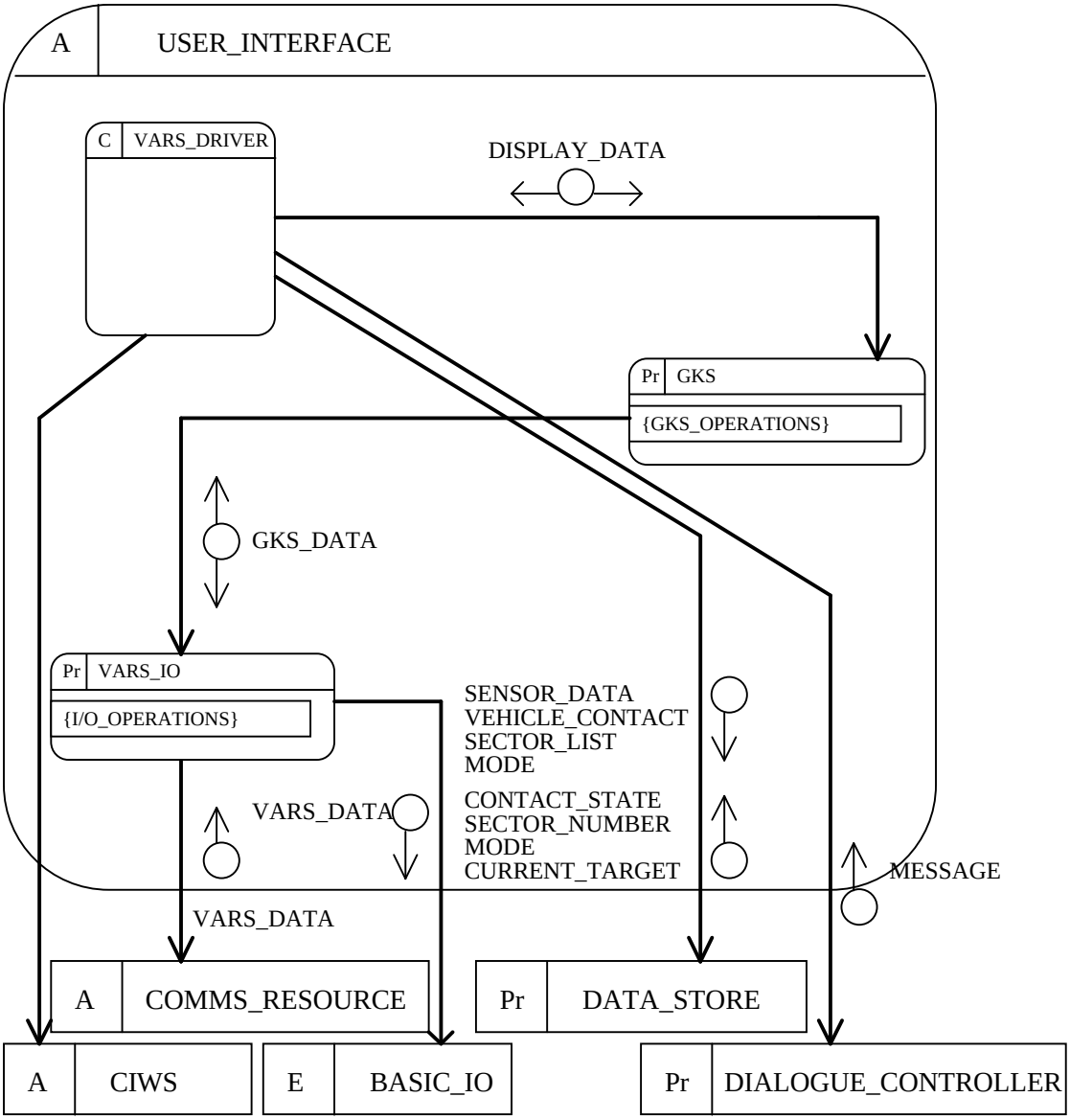

fig (7) The USER_INTERFACE object

2.7 CIWS

In many respects this object forms the core of the case study. This is the weapon system that engages incoming targets if they fill the predefined engagement criteria. Originally this object was visualised as a simple terminal abstraction of SPORADIC type, to which eligable targets were forwarded, and attacks made or aborted, by the user. However as the design evolved the CIWS was given responsibilities both to examine the DATA_STORE object for eligable targets and engage them (or abort if a user request is so received).

The complexification of responsibilities within CIWS necessitated a change of type from SPORADIC to ACTIVE, and further decomposition into four child objects (fig 8), these were:

- TARGET_SELECT (CYCLIC) - This object periodically examines the DATA_STORE object for eligable targets to engage. If an eligable target is

retrieved then it is forwarded to either CIWS_FORE or CIWS_AFT dependent on the bearing of the sensor contact from the ship.

- CIWS_FORE (SPORADIC) - This object attacks targets in the 180 degree fore arc of the vessel. When a target is received from TARGET_SELECT this object will block on the DIALOGUE_CONTROLLER TOER_PSER operation Abort_Fore_Check for *two seconds*. If an abort flag is not received from the USER_INTERFACE object within this time the CIWS will continue the attack. After a further period of *one second* the target is considered destroyed and the DATA_STORE object is updated accordingly - all further references to that contact are ignored until the end of the scenario.

- CIWS_AFT (SPORADIC) - This object attacks targets in the 180 degree aft arc of the vessel. When a target is received from TARGET_SELECT this object will block on the DIALOGUE_CONTROLLER TOER_PSER operation Abort_Aft_Check for *two seconds*. If an abort flag is not received from the USER_INTERFACE object within this time the CIWS will continue the attack. After a further period of *one second* the target is considered destroyed and the DATA_STORE object is updated accordingly - all further references to that contact are ignored until the end of the scenario.

- TIMING_LOG (PROTECTED) - This object maintains a timing log of the time taken for each abort. This is automatically dumped to the Sun3 display at the end of the scenario.

It should be noted that beyond the basic functionality outlined above the CIWS_FORE and CIWS_AFT objects also asynchronously communicate their state to the USER_INTERFACE through the DIALOGUE_CONTROLLER object. These messages are:

- ATTACKING TARGET N - DO YOU WISH TO ABORT? - prompts the user for an abort request

- ATTACKING TARGET N - informs the user that an attack against a given target has begun.

- CONTACT LOST TARGET N - the contact has been lost / destroyed

- ATTACK ON TARGET N ABORTED - the attack on the target has been aborted

It should be noted that N refers to a numbered vehicle within the scenario file.

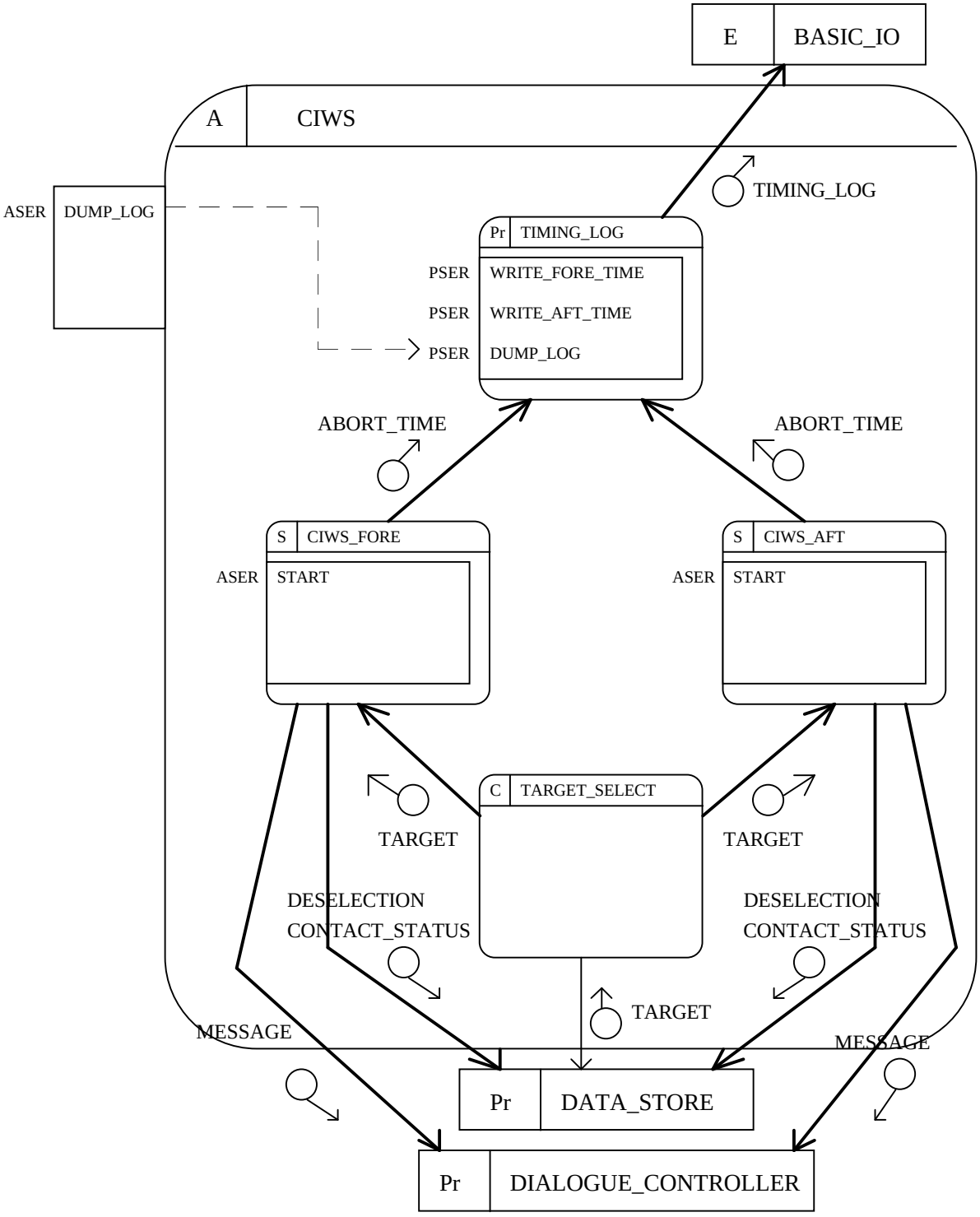

fig (8) The CIWS object

2.8 DIALOGUE_CONTROLLER

This PROTECTED object manages the interaction between the USER_INTERFACE and CIWS objects. The DIALOGUE_CONTROLLER has two key responsibilities:

1. Messages written from the CIWS for consumption by the USER_INTERFACE object (see CIWS above).

2. Requests sent from the USER_INTERFACE to abort an attack by the CIWS.

In order to satisfy these requirements DIALOGUE_CONTROLLER was decomposed into the following child configuration (fig 9).

- ABORT_CONTROLLER (PROTECTED) - This object manages the abort between the USER_INTERFACE and CIWS. CIWS blocks on either the Abort_Fore_Check or Abort_Aft_Check TOER_PSER operations for two seconds. If an abort flag is not received from USER_INTERFACE the CIWS will then continue to attack and destroy the target.

- MESSAGE_LOG (PROTECTED) - This object is a repository for messages sent from the CIWS to USER_INTERFACE (see CIWS above).

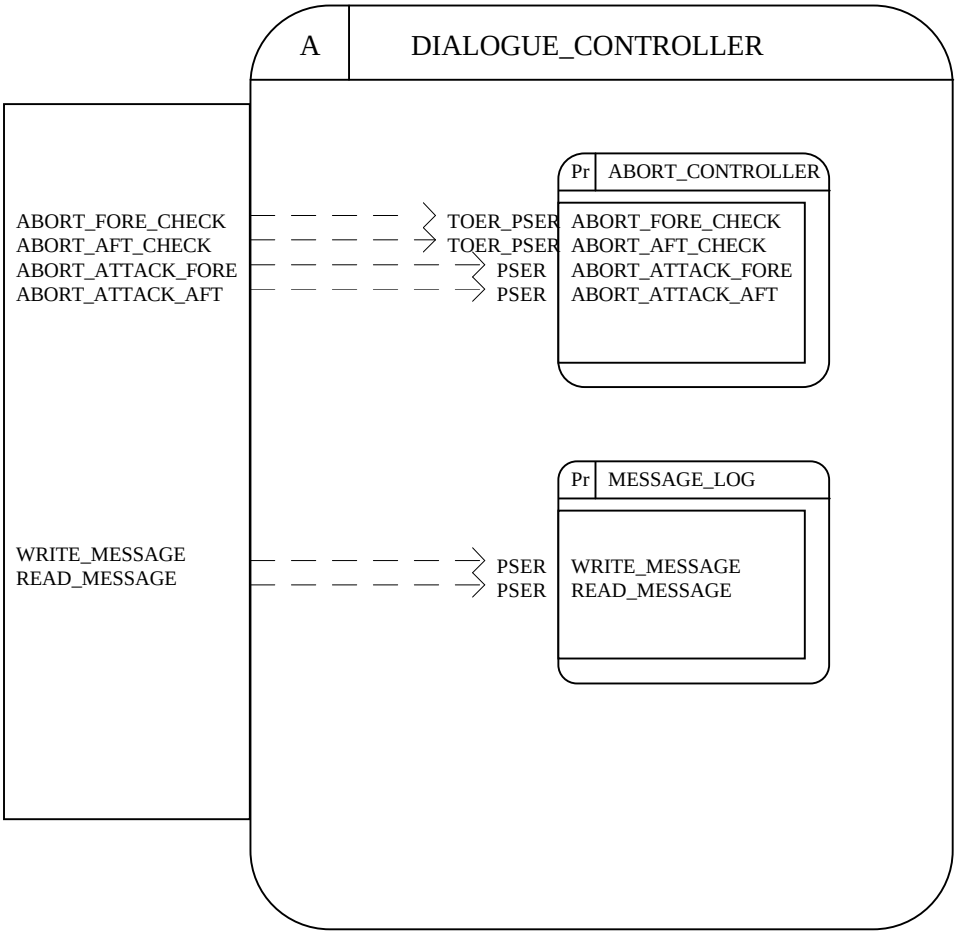

fig (9) The DIALOGUE_CONTROLLER Object

2.9 Terminal ODS Specification

As the CIWS design was to be implemented using Ada, the behaviour of each terminal object was specified using an Ada-derived pseudocode to facilitate easy translation to the target language.

In order to submit the design for timing analysis using the WCET analyser it was necessary to translate the terminal specification of each ODS into Ada code. One of the strengths of the HRT-HOOD approach is that it provides a clear and comprehendable design translation scheme for both Ada '83 and 95. This mapping scheme is briefly summarised below [Burns 1992] (fig 10):

Object	Mapping
PASSIVE	An Ada package with each operation as a procedure / function
ACTIVE	An Ada package with each operation as a procedure / function and a set of internal tasks and a synchronisation agent
PROTECTED	An Ada package with each operation as a procedure / function, and a synchronisation agent
CYCLIC	An Ada package with each ATC operation as a procedure / function, a synchronisation agent and a single periodic task. The synchronisation agent can affect the flow of control in the task.
SPORADIC	An Ada package with each ATC and START operation as a procedure / function, a synchronisation agent and a single aperiodic task. The synchronisation agent can affect the flow of control in the task. For the START operation the aperiodic task executes one iteration. Alternative mappings may be provided for SPORADICs invoked by interrupt.

fig (10) A Summary of the HRT-HOOD Mapping Scheme

One of the most consistent criticisms levelled at the Ada '83 language was the inadequate support for hard real-time development [Burns 1989]. The York/Ada 68020 compiler used as the basis for both specification and implementation of the CIWS case study addressed these issues by offering modified and extended facilities for real-time development. The modifications pertinent to the CIWS case study are outlined as follows [Firth 1992] (fig 11):

Modification	Overview
Large Priority Range	The scheduler supports 64 run queues from 0 (the default) to 63. In addition the developer may specifiy priorities in the range 64..70 to indicate hardware prioritisation.
Priority Queuing	Entry queues are now ordered according to the priority of the calling task, rather than the usual FIFO arrangement.
Protected Tasks	A monitor-like construct using the Ada '83 task semantics. Used as the basis for asynchronous dialogue between system tasks.
Delay_Until	This procedure is implemented by subtracting the current time from a later time supplied as a parameter. This will then delay the task for the appropriate amount of clock ticks (unlike the original delay statement which does not enforce an upper bound on the delay time).
Fast Interrupt Handlers	This facility allows the association of an interrupt address with a procedure, rather than the usual entry clause address attached to a task, which tends to be performance-inefficient.

Fig (11) Real-Time Modifications to the 68020 Ada Compiler

In order to constrain the design for worse case timing analysis the following limitations were placed on the Ada-based ODS specifications [Forsyth 1992] [Burns 1994b]:

- pragma LOOPCOUNT - This implementation dependent pragma is used to indicate the maximum number of times that a loop is executed.

- No Recursion - This is not permitted as it .makes it difficult for the WCET analyser to detect loops.

- No Goto - This is not permitted as it makes it difficult for the WCET analyser to detect loops.

- No Non-Static Array Operations - To implement operations on an array (such as assignment) a "hidden loop" is generated by the compiler. Non-static array operations are therefore prohibited, as the number of times a "hidden loop" is executed cannot be bounded.

3.0 The Physical Design

HRT-HOOD supports the development of a physical architecture through the association of timing and schedulability information with terminal objects. The REAL_TIME_ATTRIBUTES section of the ODS provides a framework through which the terminal set of objects in a HRT-HOOD design decomposition may be analysed to determine if they are correct with respect to their original real-time requirements.

In this section of the paper we will trace the influences the physical phase of development exerted on the final design and implementation of the case study. Like the logical design step this was very much an evolutionary process and aptly demonstrates the essentially iterative relationship between the logical and physical phases of development.

3.1 Worse Case Execution Time Analysis

Our first examination of the worse case timings revealed a number of significant problems with the logical architecture. These problems were primarily focussed on the USER_INTERFACE object. For the purposes of clarity the original worse case times for each threaded object are listed below (fig 12):

Threaded Object	WCET (ms)
COMMS_CONSTRUCTOR	295.177
VARS_DRIVER	9274.06
TARGET_SELECT	469.278
CIWS_FORE	4.25741
CIWS_AFT	4.25741

fig (12) Threaded Object Worse Case Execution Times

Obviously the WCET for the VARS_DRIVER task was wholly unacceptable, and we sought ways to reduce this significantly by examining both the logical design and the original YARD interface code (from which the USER_INTERFACE object was derived). Examination of the design indicated that VARS_DRIVER periodically invoked the following operations (fig 13):

Operations	WCET (ms)
Display_Handler.Check_Events	199.356
" . Update_Graticule	239.733
" . Update_Icons	2014.21
" . Update_Track_Display	2405.34
" . Update_Status_Line	74.2916
Gks.Set_Deferral_State	5.13167 (x2)
Gks.Redraw_All_Segments_On_WS	1.61596
Target_Handler.Update_Positions	4327.61
others	1.6401

fig (13) Operations Invoked by VARS_DRIVER

A general implementation constraint that placed an enormous overhead on the performance of the application was the protocol between the 68020 and VARS-H nodes. This dialogue was accomplished through the use of a character-based protocol, with all operations implemented by the GKS object ultimately being implemented in terms of character packets dispatched along the serial line to the VARS. The time taken to construct these commands, and the time required to flush them through the serial port imposed enormous overheads.

An obvious solution was to make the display as "minimalist" as possible, either in terms of removing extraneous features of the PPI or reducing the number of plotted vehicle icons (which took 60.1462 ms *per icon* to plot!). Close examination of the most computationally expensive operations, namely update_icons, update_track_display and update_positions showed that these iterated through a vehicle list updating each respective vehicle in turn. Our original assumption was that 50 vehicles would be present on the display at any one time. While this was not an unreasonable "real-world" estimate, the time taken to construct and dispatch the respective graphical commands required by each operation was somewhat prohibitive to performance. After consulation with the DRA this was reduced to 2 vehicles in the worse case**.

This produced the following revision of worse case times for VARS_DRIVER (fig 14):

Operations	**WCET (ms)**
Vars_Driver.Check_Events	199.356
" . Update_Graticule	239.733
" . Update_Icons	237.014
" . Update_Track_Display	283.145
" . Update_Status_Line	74.2916
Gks.Set_Deferral_State	5.13167 (x2)
Gks.Redraw_All_Segments_On_WS	1.61596
Vars_Driver.Update_Positions	505.01
others	1.6401

fig (14) Revised Threaded Object Worse Case Execution Times

While the results were still somewhat excessive, the new vehicle assumptions did significantly reduce the original WCET, bringing the VARS_DRIVER thread WCET down to 1552.07 ms (with an *average execution time* of 880 ms).

The thread WCET for TARGET_SELECT was also an area of concern. This object periodically polled the DATA_STORE repository for new targets, which required that each vehicle be compared against the eligability criteria (see section 1.1). However, with the reduction of our worse case assumption from 50 to 2 vehicles, the thread WCET shrunk from 469.278 ms to 29.181 ms.

** This represented the maximum number of missiles present in the largest scenario we had avaliable. All other vehicles (i.e. ships, helicopters, subs, planes etc) would not be plotted to the display.

3.2 Schedulability Analysis

For each thread the following overheads were added to the WCET derived from the timing analysis tool (fig 15):

Overhead	Time (ms)
Context Switch (In)	0.216
Context Switch (Out)	0.184 + 0.0048 per shorter delay task
Protected Entry (Unblocked)	0.088
Protected Entry (Blocked)	0.252
Timed Protected Entry (Blocked)	0.276
Protected Exit	0.136
Interrupt Handling Overhead	0.044

fig (15) Ada Runtime System Overheads

This resulted in the following WCET and prioritisation of the tick driven scheduler, arriving interrupts and system tasks (fig 16):

Task_ID	WCET (ms)	Priority
CLOCK	0.024	H*
VARS_INTERRUPTS	9.630**	69***
SUN_INTERRUPT	0.535**	69***
CIWS_FORE	5.293	15
CIWS_AFT	5.293	15
TARGET_SELECT	29.815	10
COMMS_RESOURCE	295.806	8
VARS_DRIVER	1562.830	2

** 'H' represents the highest priority*
*** See fig (17) Note 3 .*
**** The hardware interrupt priority (5)+ highest system priority (64)*

fig (16) The 'Task' Prioritisation Hierarchy

With the exception of VARS_DRIVER, the prioritisation hierarchy of system tasks accurately reflected the relative criticality of each individual object as derived from the original requirements.

After a comprehensive process of re-evaluating the original YARD interface implementation, the VARS_DRIVER thread WCET still remained a significant bottleneck to overall system performance. The display implementation (and the subsequent reverse engineered design), appeared to be a satisfactory *functional* design solution but was hindered by the performance constraints imposed by the underlying serial port communications infrastructure. As these constraints were specified in the original requirements they were not subject to change.

The WCET for the VARS_DRIVER thread exceeded 1.5 seconds which dictated that each update of the display would take at least this long, *assuming that the thread was released immediately after completion and ran at the highest priority*! The display update rate dictated by VARS_DRIVER was too slow to fulfill the *two*

second abort requirement, even without the obligatory interference generated by more critical tasks. The overheads imposed by generating and sending a *request abort* message to the display, the response time of the operator, and the dispatch, unpacking and actual execution of the abort far exceeds the two second requirement. While alloting the VARS_DRIVER thread a higher priority would minimise interference, the rapid update rate necessitated by the requirements would in all probability make lower priority "hard" tasks unschedulable. In other words the design would still be incorrect.

The most "satisfactory" (and schedulable) solution was to demote the VARS_DRIVER from a "hard" to "soft" level of importance, and run it at the lowest level of priority, allowing it to soak up all the spare processor capacity.

The schedulability analysis was based on the following model (fig 17):

Task_ID	Period[5]	Deadline	WCET	Block	Priority
CLOCK[1]	10.0	10.0	0.024	0	H*
CIWS_FORE_REL[2]	300.0	300.0	0.072	0	H*
CIWS_AFT_REL[2]	300.0	300.0	0.072	0	H*
TARGET_SELE_REL[2]	300.0	300.0	0.072	0	H*
COMMS_RESOURCE _REL[2]	700.0	700.0	0.072	0	H*
VARS_DRIVER_REL[2]	10000**	10000**	0.072	0	H*
VARS_INTERRUPTS[3]	103.5	-	9.63	0.288Θ	69
SUN_INTERRUPT[4]	3.3	-	0.535	0.288Θ	69
CIWS_FORE	300.0	300.0	5.293	30.778	15
CIWS_AFT	300.0	300.0	5.293	30.778	15
TARGET_SELECT	300.0	300.0	29.815	30.778	10
COMMS_RESOURCE	700.0	700.0	295.806	30.778	8
VARS_DRIVER	10000**	10000**	1562.83	0	2

**H - represents the highest system priority*
*** - An arbitary "soft" value*
Θ - *this represents the maximum time the Sun and Vars interrupt were turned off.*

fig (17) The Schedulability Analysis

Notes

(1) This models the clock handler, with a "tick" of 10ms and an overhead for recognising pre-emption of 24μs.

(2) Each _REL "task" models the interference generated by the scheduler by moving each respective system task from the delay queue (72μs).

(3) The arrival rate of interrupts from the VARS was "bursty" with a worse case arrival of 18@440μs intervals and a gap of 95.6 ms. This combined value (plus overheads) of 18 * 440μs was utilised to model the interference due to interrupt handling on lower priority tasks.

(4) The worse case arrival rate was a constant dictated by the speed of the serial link (set at 2400 BAUD). This constant was utilised to calculate the listed period.

(5) For all other tasks period *equals* deadline.

From the schedulability analysis we were able to determine the following response times for the CIWS case study (fig 18).

Task_ID	Response Time (ms)
CLOCK	0.256*
VARS_INTERRUPTS	11.068**
SUN_INTERRUPT	11.068**
CIWS_FORE	55.3
CIWS_AFT	61.68
TARGET_SELECT	97.459
COMMS_RESOURCE	557.501
VARS_DRIVER	8935.872(!)

**This was calculated as follows: the response time of the clock (0.024ms) + the release time of the first task (0.072ms), plus release time for all four other tasks (4 * 0.040ms).*
*** As interrupt arrivals were "bursty" this led to a seperate determination of response time, with the longest being 11.068 ms.*

fig(18) System Response Times

3.3 An Alternative Design

Although the full system is incorrect with respect to its original non-functional requirements, we were able to produce a "correct", albeit limited, version of the CIWS system that met the key abort criteria. The requirements were changed in the following manner:

- The system was restricted to a Sun3 and 68020 configuration.
- The user interacted with the system through a simple text interface on the Sun3.
- The only commands accepted by the system were (a) abort an attack by the fore gun (b) abort an attack by the aft gun.
- The system would operate in AUTOMATIC mode only.
- Only messages indicating the state of the CIWS were echoed to the Sun3. The original functionality of the VARS display was not supported.

The logical architecture was adapted in the following fashion (fig 20):

- The USER_INTERFACE object was removed from the design.

- A Use relationship was established between COMMS_MANAGER and DIALOGUE_CONTROLLER - this allowed the user to abort an attack from the Sun3 node through a simple text interface.

- A Use relationship was established between MESSAGE_LOG and the environment object BASIC_IO. This allowed messages from the CIWS to be output to the Sun3 node.

The breakdown of system tasks was as follows (fig 19):

Task_ID	WCET (ms)	Priority
CLOCK	0.024	H*
SUN_INTERRUPT	0.535	69**
CIWS_FORE	5.293	15
CIWS_AFT	5.293	15
TARGET_SELECT	194.09	10
COMMS_RESOURCE	295.806	8

** 'H' represents the highest priority*
*** The hardware interrupt priority (5)+ highest system priority (64)*

fig 19 The "Alternative" Prioritisation Hierarchy

The removal of the USER_INTERFACE object from the logical architecture facilitated the following changes to the analysis model:

- The worse case assumption of tasks moved by the scheduler from the delay queue could be reduced from 5 to 4 (which consequently reduced the clock response time in the worse case).

- The two node architecture facilitated the removal of the VARS_INTERRUPTS from the analysis.

- The worse case number of targets could be increased from 2 to 20.

- The maximum blocking time (originally caused by VARS_DRIVER) was reduced from 30.778ms to 26.576ms.

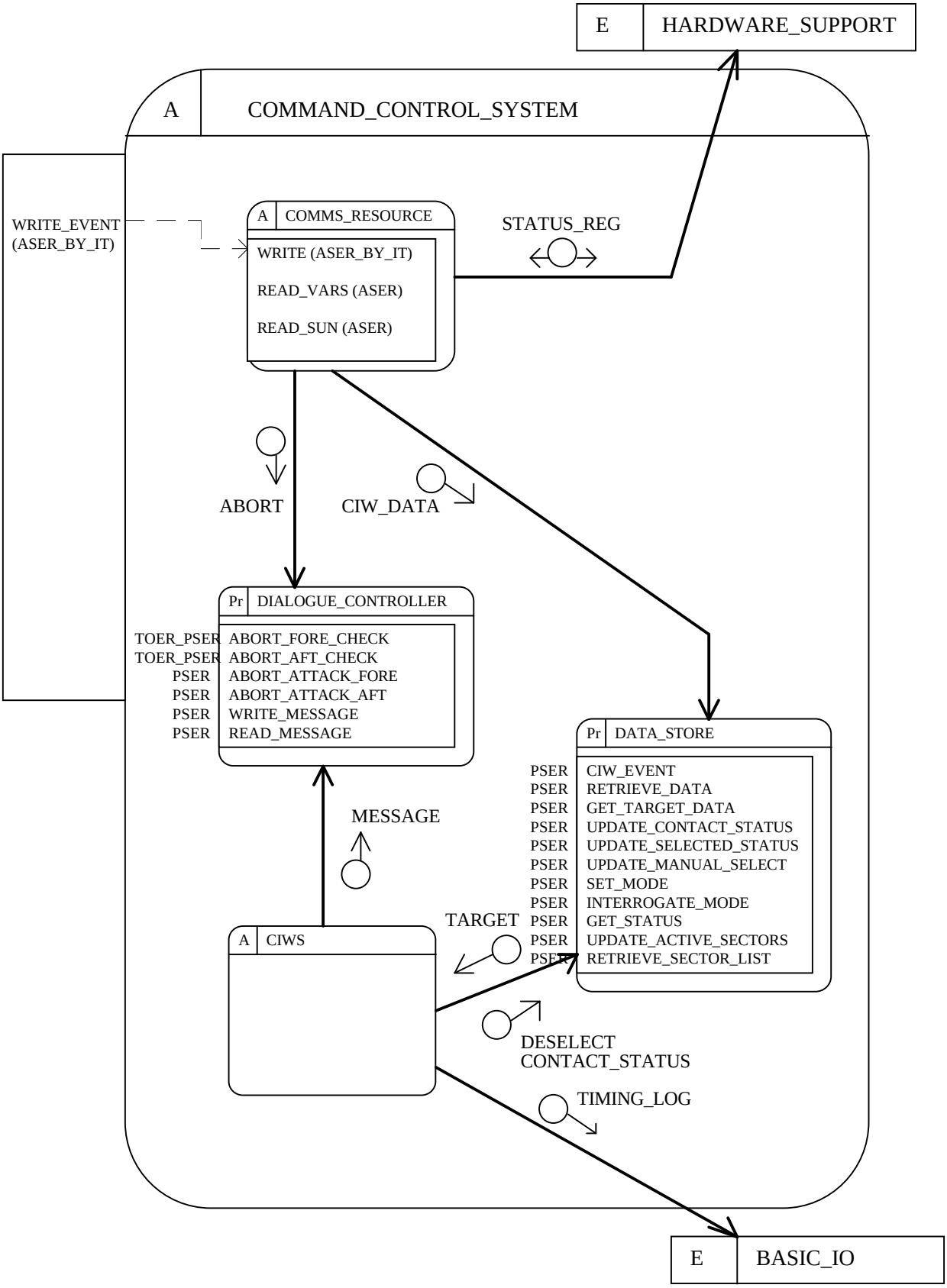

fig (20) The "alternative" design

The analysis included the following components (fig 21):

Task_ID	Period[4]	Deadline	WCET	Block	Priority
CLOCK[1]	10.0	10.0	0.024	0	H*
CIWS_FORE_REL[2]	700.0	700.0	0.072	0	H*
CIWS_AFT_REL[2]	700.0	700.0	0.072	0	H*
TARGET_SELE_REL[2]	700.0	700.0	0.072	0	H*
COMMS_RESOURCE _REL[2]	600.0	600.0	0.072	0	H*
SUN_INTERRUPT[3]	3.3	-	0.535	0.288Θ	69
CIWS_FORE	700.0	700.0	5.293	26.576	15
CIWS_AFT	700.0	700.0	5.293	26.576	15
TARGET_SELECT	700.0	700.0	194.09	26.576	10
COMMS_RESOURCE	600.0	600.0	295.806	0	8

**H - represents the highest system priority.*

Θ *- this represents the maximum time the Sun and Vars interrupts were turned off.*

fig (21) The "Alternative" Schedulability Analysis.

Notes

(1) This models the clock handler, with a "tick" of 10ms and an overhead for recognising pre-emption of 24μs.

(2) Each _REL "task" models the interference generated by the scheduler by moving each respective system task from the delay queue (72μs).

(3) The worse case arrival rate was a constant dictated by the speed of the serial link (set at 2400 BAUD). This constant was utilised to calculate the listed period.

(4) For all tasks period *equals* deadline.

From the schedulability analysis we were able to determine the following response times for the alternative CIWS design (fig 22).

Task_ID	Response Time (ms)
CLOCK	0.216*
SUN_INTERRUPT	11.350
CIWS_FORE	38.673
CIWS_AFT	45.060
TARGET_SELECT	277.152
COMMS_RESOURCE	599.580

**This was calculated as follows: the response time of the clock (0.024ms) + the release time of the first task (0.072ms), plus release time for all four other tasks (3 * 0.040ms).*

fig (22) "Alternative" System Response Times

3.4 Results

Through the analysis process outlined above, and rigorous testing of the final implementation we were able to determine two key points of interest concerning performance:

1. The maximum number of targets that the CIWS can destroy in AUTOMATIC mode.

2. The shortest abort time recorded by the TIMING_LOG object.

For (1) we made the assumption that the worse case number of targets are fired at the ship, and simultaneously reach the 5000 metre engagement range. The system is left in AUTOMATIC mode, and will engage these targets in turn, waiting for 2 seconds to request an abort, then delaying a further second to destroy the target. No abort is assumed to be received by the system.

The worse case attack time is simply the end-to-end transaction time derived by adding together the periods of COMMS_RESOURCE , TARGET_SELECT and either CIWS_FORE or CIWS_AFT. The following table lists the maximum number of targets that can be destroyed by the full and alternative designs, assuming the worse case scenario outlined above (fig 23):

Closing Speed[1]	**Time To Target**[2]	**Full Design**[3]	**Alt. Design**[4]
200 m/s	25.0 s	2	5
300 m/s	16.66 s	2	3
400 m/s	12.5 s	2	2
500 m/s	10.0 s	2	2

fig (23) Targets Destroyed In Worse Case Scenario

Notes:

(1) Closing speed on the ship, measured in metres per second.

(2) The number of seconds required for the missile to move from maximum engagement range to impact.

(3) This column lists the number of missiles that may be destroyed by the full design (with the VARS display) in the worse case scenario. The worse case number of targets is 2.

(4) This column lists the number of missiles that may be destroyed by the alternative design (without the VARS display) in the worse case scenario. The worse case number of targets is 20.

When viewing these results it should be noted that for *each* target engaged by the system there is a *three second overhead* specified by the requirements; two seconds to wait for a possible abort request by the user, and a further second to attack and destroy

the selected target. These overheads have a significant impact on the worse case attack time, for example a system that ran *instantaneously* would only be able to engage eight targets travelling at 200 m/s , five targets travelling at 300 m/s and so on.

The shortest recorded abort time for both the full and alternative designs are listed below. Note that these times are best case results, that also include a period of 'thinking time' by the user (fig 24).

Design	Abort Time (seconds)
Full Design	**2.6**
Alternative Design	**0.9**

fig (24) Best Case Abort Times

4.0 Recommendations

It is our opinion that the strict performance constraints imposed by the communications infrastructure placed serious overheads on the CIWS design. As we have demonstrated, the primary bottleneck appeared to be in the USER_INTERFACE object which updated the display and system state in response to contacts recieved by the CIWS or commands issued by the user. Primarily this performance shortfall (in order of precedence) was due to:

- The serial-based communications architecture, particularly in the transmission of control characters from the 68020 board (through an MC68681 serial controller) to the VARS.

- The complex conversion process required to convert the design and implementation based data structures used by the CIWS into commands for the VARS display (accomplished by the VARS_IO object).

Our primary recommendation would therefore be to seriously reconsider the original hardware constraints specified in the requirements document. The following list encompasses tentative (and mutually exclusive) possibilities for change:

- A fast ethernet-based communications infrastructure between the 68020 board and the VARS-H display.

- A dedicated processor for driving the VARS-H display.

- Replacing the 68020 / VARS-H configuration with a single processor board with dedicated graphics hardware, display and input device.

As we have shown through the “original” CIWS design (presented above), the current hardware architecture is insufficient to meet the hard deadlines specified in the requirements, particularly the “abort” function. As the “alternative” design demonstrates, the removal / amelioration of these constraints would allow the formulation of a hard real-time system “correct” with respect both to its functional and temporal requirements.

5.0 Conclusions

In this study we have traced the design and analysis of a simple command-control system using the HRT-HOOD development technique. HRT-HOOD offers a flexible, comprehendable and expressive hierarchical design notation for object-oriented real-time development while still maintaining a manageable degree of simplicity and ease-of-use. Fundamentally, HRT-HOOD offers the developer leeway to consider the implication of environmental constraints on performance at the design level, allowing the design to be easily and inexpensively changed if the logical architecture does not meet its required non-functional obligations. This is in direct contrast to the more traditional view of real-time development, where non-functional obligations are considered at the implementation stage of the lifecycle, and a necessary change to the design becomes more expensive by several degrees of magnitude. Finally the transition from design to implementation, often ill-defined in many structural notations, is supported by a simple (and automatable) direct structural mapping to Ada '83 and 95 source code.

Through the CIWS case study we have examined both the logical and physical aspects of real-time development using HRT-HOOD. By tracing the structural evolution of the logical architecture we have demonstrated how hierarchical structuring and object-based modularity using HRT-HOOD facilitates the development of comprehendable design solutions. Enhanced comprehendability facilitates not just easy communication of the design solution to a third party, but also has the potential to significantly reduce maintenance costs in the long term.

Our physical analysis of the CIWS design aptly demonstrated the advantages of considering performance requirements at the design level. The failure of the CIWS to meet performance requirements (due to the overheads imposed by the serial communications architecture) was identified at the detailed design stage. This facilitated a review of the specified hardware constraints, the formulation of a set of recommendations for change, and the development of an "alternative" design that demonstrated the correctness of the design without the overheads imposed by the VARS display. An implementation-centred approach to analysis would potentially have been more costly.

References

Audsley 1993a
N.C. Audsley, A.Burns and A.J. Wellings 1993 "Deadline Monotonic Scheduling Theory and Practice" , Control Engineering Practice, Vol 1, No. 1, Pages 71-78.

Audsley 1993b
N.C.Audsley, A. Burns, M. Richardson, K,Tindell and A. Wellings 1993 “Applying New Scheduling Theory To Static Priority Pre-emptive Scheduling”, Software Engineering Journal, Vol 8, No 5.

Burns 1989
A. Burns and A.J. Wellings. 1989 "Ada 9X - The Need For Change". YCS 117 [Internal Report].

Burns 1992
A. Burns and A.J. Wellings 1992 "HRT-HOOD: A Design Method for Hard Real-Time Ada 9X Systems", Proceedings of Ada UK 1991.

Burns 1993a
A. Burns, A.J. Wellings, C.M. Bailey and E.Fyfe 1993 "The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-Time System Design and Implementation". YCS 190 [Internal Report]

Burns 1993b
A. Burns and A.J. Wellings 1993 “Bridging the Real-Time Gap between Ada’83 and Ada 9X”, in E.C. Loftus (ed) 1993 Ada Year Book. IOS Press.

Burns 1994a
A. Burns and A.J. Wellings 1994 "HRT-HOOD: A Design Method for Hard Real-Time Ada", Real-Time Systems Journal, Vol 6, pages 73-114.

Burns 1994b
A. Burns, A.J. Wellings, C.H. Forsyth and C.M. Bailey 1994 "A Performance Analysis of a Hard Real-Time System". YCS 224 [Internal Report]

Cornwell 1993
Pete Cornwell and Andy Wellings 1993 "The Close-In Weapons System, A Case Study in HRT-HOOD (Preliminary Design)". DRA Deliverable.

Firth 1988
J.R. Firth, C.H. Forsyth and I.C. Wand 1988 "York Ada Compiler Release 3 (SUN/UNIX) - User Guide". YCS 97 [Internal Report]

Firth 1992
John Firth 1992 "Hard Real-Time Operating System Kernel Study, Task 8, Volume A, Modifications to York Ada", ESTEC/Contract No. 9198/90/Nl/SF. York Software Engineering Limited.

Forsyth 1992
C.H. Forsyth 1992 "Hard Real-Time Operating System Kernel Study, Task 8, Volume E, Implementation of the Worse-Case Execution Time Analyser", ESTEC/Contract No. 9198/90/Nl/SF. York Software Engineering Limited.

Glen 1992
J.A. Glen 1992 "Statement of Requirements for a Feasibility Study of Rigorous Development Methods for Real-Time Software-Intensive Projects". Yard Document 4808, Contract No. C2432, Yard Ltd Consulting Engineers, Glasgow.

Hutcheon 1992
A.D. Hutcheon 1992 "Hard Real-Time Operating System Kernel Study, Task 8, Volume C, Timings of Run-Time Operations in Modified York Ada", ESTEC/Contract No. 9198/90/Nl/SF. York Software Engineering Limited.

Lister 1990
A. Lister and A.Burns 1994 "An Architectural Framework for Timely and Reliable Distributed Information Systems (TARDIS): Description and Case Study". YCS 140 [Internal Report]