Hard Real-Time Software Tools

and Programming Languages Environment

Version 2.0

Prepared by Yves Meyland at ESTEC/WSD
Supervised by Tullio Vardanega (WSD)

August 12, 1997

Table of Contents

1. INTRODUCTION

1.1 Purpose

This document reports the experience gained from the usage of a complete set of tools aimed to support the development of hard real-time software. The complete process cycle started through research and development activities which led to the industrialisation contract of the ERC32 computer system and associated development environment. My main tasks were to assess, integrate, and demonstrate the feasibility of using specific tools for the development of hard real-time software. Therefore, this document is for the user to understand the relationships and dependencies between all the different tools and languages, and how they interact with one another. In the longer term, this technology will be used for future projects for the development of on-board software such as the PROBA satellite. The last section of this document summarizes my activities spent using this technology for the part of the Onboard Operations Support Software (OBOSS).

1.2 Scope

This document is restricted to a top level view of the software tools. Its intention is not to be a second user manual for each tool vendor but to demonstrate the relationships and interactions between them. It shows the process flow of the software development for on-board space application.

1.3 Structure

Section 2 presents all the different tools used for the development of hard real-time software for on-board space application.

Section 3 identifies the programming languages used and their particular features.

Section 4 gives a personal assessment of the overall process.

Section 5 summarizes my work done on OBOSS.

Section 6 concludes the report.

Appendix A describes a specific example of a software model presented at DASIA'97.

Appendix B shows some diagrams of the output of the various tools.

1.4 Reference Documents

[R1] Spacebel Informatique
Target Simulator
32-Bit Microprocessor and Computer System Development
User's Guide
32B-SBI-SUM-0189-003, Issue 2 Revision 1, 19/12/96

[R2] Spacebel Informatique
Schedulability Analyser and Scheduler Simulator
32-Bit Microprocessor and Computer System Development
Software User Manual
32B-SBI-SUM-0189-001/2, Issue1 Revision 3, 20/12/96

[R3] Thomson Software Products
RISC Architecture and Technology
Worst Case Execution Time Processing
Preliminary User's Guide
RAT-ALS-WP20-PUG-001, Issue 2 Revision 0, 29/11/96

[R4] Thomson Software Products
Ada Compiler Modification
32-Bit Microprocessor and Computer System Development
MCD-ALS-P2-DS-001, Number 01 Version 2.1, 29/11/96

[R5] Thomson Software Products
Ada Runtime System Modifications
32-Bit Microprocessor and Computer System Development
MCD-ALS-P2-DS-003, Number 03 Version 2.1, 29/11/96

[R6] Yves Meylan - European Space Agency (ESA) / ESTEC
Communication Mechanism Handling and Mapping Translation
between HRT-HOOD and SDL
Version 3.0, 14/04/97

[R7] Yves Meylan - European Space Agency (ESA) / ESTEC
Ada Template Rule for ERC32 Code Extraction
Version 2.0, 21/02/97

2. Hard Real-Time Software Development Tools

2.1 Introduction

There are four main different tools and methodologies used in the described software development life cycle:

1 - Hard Real-Time Hierarchical Object Oriented Design (HRT-HOOD)

2 - ERC32 Cross Ada Compilation System (ACS)

3 - Schedulability Analyzer, Simulator, and Target Simulator

4 - Specification and Description Language (SDL)

Each tool interacts in a direct or indirect way with the other. Figure 2.1.1 illustrates the various interactions and relationships between all tools except for SDL. The specification, with the use of SDL, occurs in the first step of the software life cycle. Following the user requirements, the model hierarchy and data are inserted into the SDL environment. Some of the defined objects are then transferred in the HRT-HOOD environment while others are refined in SDL for the system requirement phase. Figure 2.1.2 describes this process flow. The complete details of all the interactions are part of the next sections of this report.

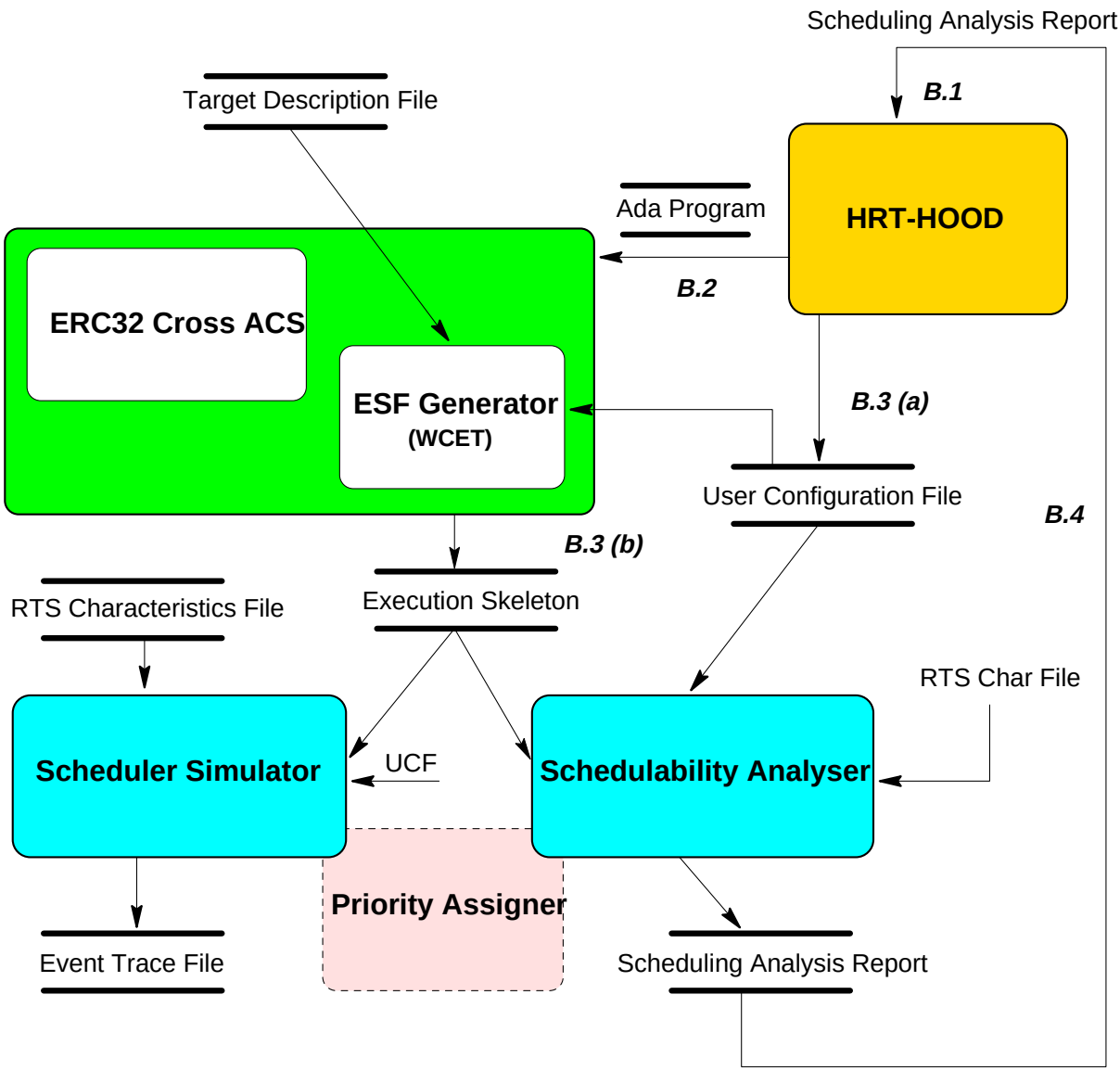
Scheduling Analysis Report
Target Description File
B.1
HRT-HOOD
Ada Program
B.2
ERC32 Cross ACS
ESF Generator
(WCET)
B.3 (a)
User Configuration File
B.4
B.3 (b)
RTS Characteristics File
Execution Skeleton
RTS Char File
Scheduler Simulator
UCF
Schedulability Analyser
Priority Assigner
Event Trace File
Scheduling Analysis Report

Figure 2.1.1

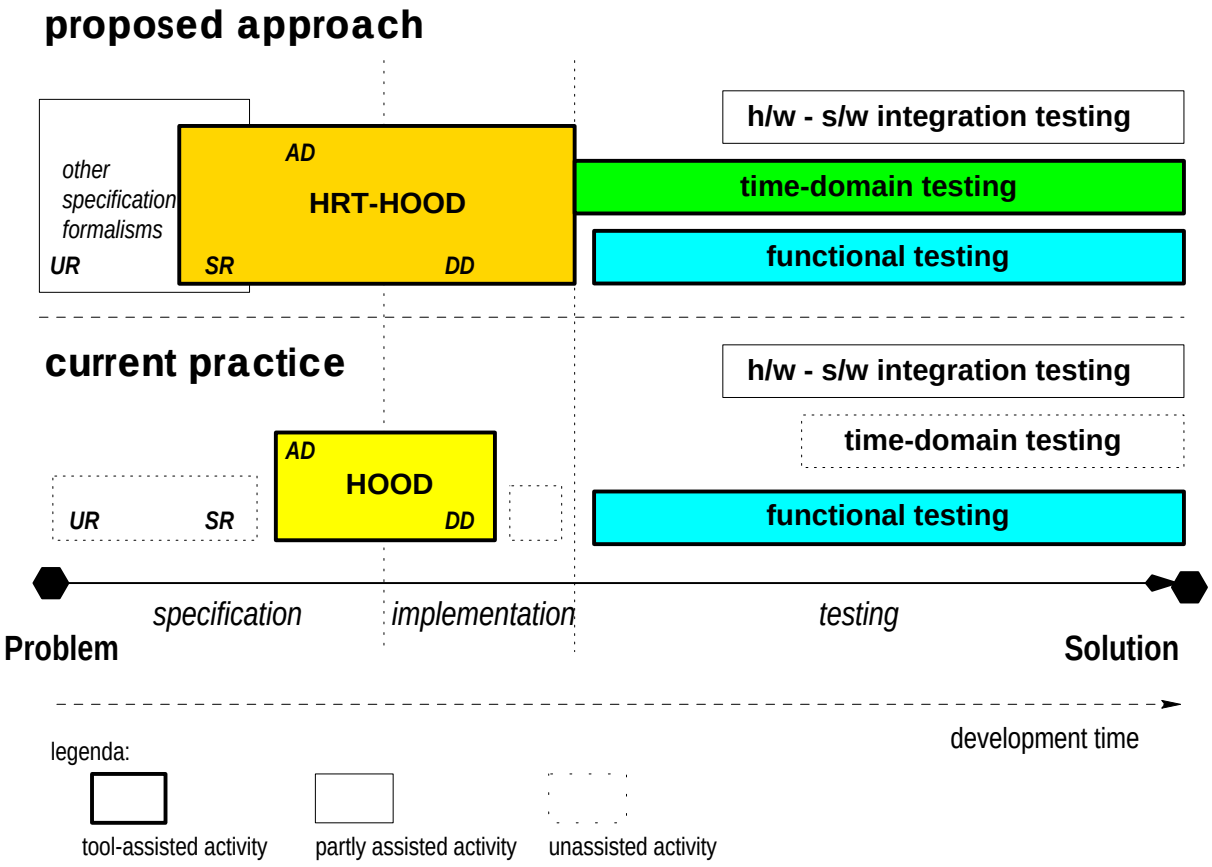

Figure 2.1.2

2.2 Hard Real-Time Hierarchical Object Oriented Design (HRT-HOOD)

The HRT-HOOD tool ***HoodNICE*** provided by Intecs Sistemi is intended for the design phase of the software life cycle of a system. It is to be designed for a hard real time environment. The designer uses an object oriented methodology by using objects of type cyclic, software sporadic, interrupt sporadic, protected, active, and passive. An example of a HRT-HOOD diagram can be found in Appendix B, Figure 1.

All these objects, except for the passive, have specific timing constraints and real-time operation attached to them. Various timing attributes can be defined, such as the period, inter-arrival time, deadline, depending on the type of object. Appendix B, Figure 2, shows an example for a cyclic object's real-time attributes.

All the data and information inserted into the tool model can be used to extract Ada code which includes some Ada95 real-time features. The extraction is obtained by executing the following commands:

1 - Mouse selection of the parent or child object

2 - On the HD Editor menu bar: Tools -> Code Extractor -> Recursive/Non-recursive

This process of generating code directly from the design model is the first essential step for interacting with the other software tools used in the overall process flow. The extracted code has some specific features which includes a format known as the worst case execution time (WCET). Specific Ada WCET coding rules have to be implemented which are part of the rules

in the extraction code. These rules are described in more detail in chapter 3.1 of this report.

The HRT-HOOD tool also allows the possibility to generate a timing data file (or **U**ser **C**onfiguration **F**ile - UCF) which will be then used by two tools later in the process. This file is generated in the following way:

1 - The RTA object is part of the model as an environment object

2 - On the HD Editor menu bar: HRT -> Schedulability Analyser

3 - Select the desired operational mode

4 - Save all the timing ODS templates for all the objects part of the system

(Timing ODS -> Template)

5 - Select all the objects in the Schedulability Analyser Mode window and from the

menu bar: General -> Build Timing Data File

This file contains the real-time characteristics for each object thread defined in the model as specified in the object description skeleton. This information is necessary for later determining for instance if a task is schedulable or to analyze specific response times. An example of the contents of a UCF is shown below, and how the information is used, is described in chapter 2.4.

UCF File example:

```
PREFERENCES
   DMS
   BLOCKING PROTOCOL IPCI
END

TARGET CHARACTERISTICS
   NO ATAC
   PRIORITY LOW  1
   PRIORITY HIGH 64
   CPU CLOCK FREQUENCY 10 MHz
   WAIT STATES READ 0 WRITE 0
END

THREAD DEFINITION

THREAD   Interrupt_Sporadic.OBCS
  MINIMUM      500000
END -- Interrupt_Sporadic

THREAD   SW_Sporadic_C.THREAD
```

CRITICALITY soft
MINIMUM 170000
DEADLINE 170000
END -- SW_Sporadic_C

THREAD Cyclic_Handler.THREAD
CRITICALITY soft
PERIOD 500000
DEADLINE 220000
END -- Cyclic_Handler

WCET DATA ignore
END

Each object type has specific timing data associated with its behavior. The cyclic object has a defined period and the sporadic object has a minimum inter-arrival time. All objects except the interrupt sporadic have an identifier for its criticality and deadline. All values are expressed in clock cycle (10 MHz). More information can be obtained in [R2].

Another important feature of the HRT-HOOD tool is the ability to check the completeness and correctness of the HRT-HOOD model. This particular check in the design phase of the model ensures that the important hard real-time rules and principles are enforced in the initial phases of the software development of critical on-board applications. This check is done by selecting the HD Editor menu bar with the HRT -> HRT Checker options.

As mentioned previously, Ada code can be extracted from the design model. The code extracting templates can also be modified in order to enforce new code guidelines. It is necessary to generate a new extractor executable after updating the ASCII template file (i.e. *TEMPLATE_ACE.att_config*), in the following way:

hood_getmpl_off Ascii_Extractor_Template_File_Name Executable_Extractor_File_Name

The new executable has then to be put in the appropriate directory of the HoodNICE tool and replace the existing file (~/HoodNICE2_3_3/config*/TEMPLATE_EXE_ACE.att_config*). The Ada code of a model can also be extracted without using the GUI tool. The following command line can be given:

hood_code_extract -CTEMPLATE_EXE_ACE.att_config -R Model_Name

To summarize the important data generated from HRT-HOOD, the Ada code extracted and the timing data file are the two main pieces of information which are passed along in the next phases of the process.

2.3 ERC32 Cross Ada Compiler System

The tools provided by Aonix (formerly Alsys Inc.) are used for compiling, binding, and linking the Ada code extracted by the HRT-HOOD tool. All the phases of the compilation up to the generation of an executable file can be performed through one automated set of commands - the make (i.e. AdaMake) tool of the Ada Alsys environment. AdaMake automatically brings an

application up to date.

Various options and flags can be set for the compilation, bind, and link phases. One of these options relates to the WCET as described previously in chapter 2.2. By having the compile option *hrt_data* set to yes (i.e. *hrt_data=yes*), the compiler will perform WCET processing and store the results in the program library. In the bind and link phase, three other options are also set (*level=bind, wcete=yes, skeleton=automatic*). An example of a bind invoke file is shown below:

```
def.bind level=bind
def.bind wcete=yes
def.bind skeleton=automatic
def.bind out=auto data=all
def.bind directive=$HOME/ERC32_System_Env/pgmdem32.cmd
def.bind segment_symbols=yes
def.bind format=srecord

bind (program => Main_System,
      library => library,
      interface =>
           (modules => "$HOME/tools/ada/tsp/brd/dem32/userdem32.o"));
```

After having executed these steps error-free, an **E**xecution **S**keleton **F**ile (ESF) is generated. This file contains the definition of the execution skeleton profiles for threads and protected objects defined in the application. It gives a full description of the worst case path of a thread as a sequence of worst execution time for CPU instruction execution time, calls to protected objects, and loop statements [R2]. The user can increase the work load (i.e. increase the number of clock cycles for a particular activity within a thread) by having some back ground operations being performed such as a bounded loop structure. An ESF file example is given below:

```
PROGRAM MAIN_SYSTEM
  ...
  THREAD SW_SPORADIC_INT.THREAD
    TYPE SPORADIC
    WCET 3, 0, 0
    CALL_PO SW_SPORADIC_INT.OBCS WAIT_START
    WCET 29, 3, 4
    CALL_PO TABLE_HANDLER.OBCS WRITE_TABLE
    WCET 3, 0, 0
    CALL_PO STORE.OBCS WRITE
    WCET 4, 0, 0
  END
```

```
  THREAD SW_SPORADIC_C.THREAD
    TYPE SPORADIC
    WCET 3, 0, 0
    CALL_PO SW_SPORADIC_C.OBCS WAIT_START
    WCET 34, 4, 5
    CALL_PO TABLE_HANDLER.OBCS READ_TABLE
   END
 THREAD INTERRUPT_SPORADIC.OBCS
    TYPE INTERRUPT SPORADIC
    WCET 29, 3, 2
    CALL_PO SW_SPORADIC_INT.OBCS START
    WCET 4, 0, 0
  END
  PROTECTED BUFFER.OBCS
    TYPE RESOURCE
    ENTRY READ
      WCET 85, 8, 4
    ENTRY WRITE
      WCET 86, 7, 5
  END
...

END
```

The Alsys tools are also used to interface with the C programming language. After having modified the necessary Ada specification and/or body files in order to be able to encapsulate C pragma interface calls, the C file needs to be compiled to generate an object file (.o) for the appropriate platform (ERC32). The Microtec Research MCCSP ANSI C Compiler is used to generate the object file in the following way: *$USR_MRI/bin/mccsp -c file_name.c,* where *USR_MRI* is the environment variable specifying where the Microtec Research files are located. The Ada bind invoke file has to be then given the appropriate C interface module name so that the linker recognizes the external defined procedures and/or functions (i.e. *interface => modules ("c_file_name.o*")). Chapter 3.1 gives an Ada example with C interface.

It is possible for the user to define Ada procedures and functions which are referenced directly through their address (i.e System.Address). The package *Indirect_Call* provides a mechanism for calling any Ada procedure P indirectly via P'Address with or without procedures [R5]. The same type of indirect call can be done also with C procedures with the pragma interface. Refer to 3.1 for a complete Ada example.

The user can also write data at a specific memory location (e.g. *for Val use at #1600000000#*). Special caution should be made when assigning data to a specific memory location. The assigned location is defined during the bind and link phase, and it appears in the *user data* location of the binder output described in the next paragraph.

An important piece of reference generated from the Alsys tools comes from the binder output

which produces a *.bnd* file summarizing for instance all the user defined procedures, function, calls to run time system, C interface function allocation, status of compiled units and libraries. In order to get all the necessary information, the binder should have the option segment_symbols set on (i.e *def.bind segment_symbols=yes*).

To summarize the important data generated from the Alsys tools, the ESF, the executable system model and *bnd* files are the main pieces of information which are used to be passed along in the next phases of the process.

2.4 Schedulability Analyzer, Schedulability Simulator, Target Simulator

The use of the three software tools provided by Spacebel Informatique can be grouped into two categories. The schedulability analyzer and simulator are part of one group and the target simulator is in the second one.

2.4.1 Schedulability Analyzer and Simulator

The output of these two tools provide the designer with an accurate schedulability analysis and simulation of the model designed in HRT-HOOD. The results obtained from the tools allows the user to optimize the code or review again the design [R2].

The analyzer and simulator require three different input files - the ESF, UCF, and RTS files. Two of the three files come from the use of the previously described tools - the use of the Alsys tools to generate the ESF and the HRT-HOOD tool to generate the UCF. These two files are directly used along with a specific **R**un-**T**ime **S**ystem metrics file (RTS) containing the system overhead incurred by the computational model [R2]. An example of a RTS file is given below:

```
-- RTS Characteristics for DEM32 at 10 MHz (non-ATAC runtime)
-- Characteristics are in Processor Cycles

ENTER PASSIVE TASK                  80,0,0
EXIT PASSIVE TASK                       110,0,0
ENTER SOFTWARE SPORADIC WAIT        (70,0,0)
EXIT SOFTWARE SPORADIC WAIT         30,0,0
ENTRY QUEUE SERVICING               (60,0,0)
CONTEXT SWITCH TIME                 340,0,0
SCHEDULER SELECT TIME               (50,0,0)
ENTER DELAY UNTIL OVERHEAD AT HEAD    (390,0,0)
ENTER DELAY UNTIL OVERHEAD NOT AT HEAD (220,0,0) + (30,0,0) * C
EXIT DELAY UNTIL OVERHEAD               80,0,0
TIMER INTERRUPT OVERHEAD                210,0,0
READY AFTER DELAY                   120,0,0
INTERRUPT HANDLING OVERHEAD         670,0,0
ENTER INTERRUPT SPORADIC WAIT       (30,0,0)
EXIT INTERRUPT SPORADIC WAIT            30,0,0
MAXIMUM NON PREEMPTION                  1300,0,0
```

The results of the schedulability analyzer provide important data on the behavior of the model originally designed in HRT-HOOD. The most important piece of information is to know if a thread is schedulable or not. The tool assigns priorities to the different threads and protected objects based on the ESF, UCF, and RTS data files. It also provided additional information on the worst case computational time (WCCT), response times, task blocking origins, margin analysis (i.e. maximum WCCT not causing missed deadlines), and overall utilization factor. The UCF file needs to be modified in the case of behavior errors in the simulation. For example, it may be necessary to correct a deadline which is missed. Any modification of the timing values in the UCF requires an update of the values in the HRT-HOOD model. An example of the output of the schedulability analyzer is given in Appendix B, Figure number 3.

The output of the simulator provides different types of results. The user is able to simulate the execution of the different thread(s) and access to the protected object(s). The initial output shows the on-line trace. The first error messages can be ignored since there are only related to the first activations of the threads. However, if there are a number of additional errors which are displayed, it shows an error in the timing values of the model's real time attributes. The Gant report displays in a graphical manner the various time frames of the threads of the system. This view gives a better perspective of the behavior of the system.

The schedulability analyzer and simulator should be used in common in order to complement each other. When modifications have been performed to the UCF, both tools should be re-executed again to guarantee that the updated results are still meaningful between the two tools.

2.4.2 Target Simulator

This tool allows to simulate a computer which is built around the ERC32.

The target simulator is used to execute the .x file generated by the bind and link process and is intended for debug and simulation purposes. An important feature is the possibility to trace the execution of a program in order to debug the system - through assembly language instructions. The setting of breakpoints in hexadecimal format (*e.g. m s b 0x<hex address>*) at specific procedure address calls allows the user to stop at pre-defined points during the execution of the program in order to know in which thread the program has passed, failed, or not been reached. Two other commands are particularly useful. The first one is to schedule a Direct Memory Access (DMA) - read or write. The other command is the generation of an interrupt. When an interrupt sporadic has been defined in the HRT-HOOD model, its arrival can be simulated through the target simulator with the *interrupt* command. Note that the interrupt identification number defined in the HRT-HOOD model has a different ID than the one used on the target simulator. Refer to chapter 3.1 for more information on the interrupt ID value assignment. The target simulator causes an interrupt with the signal level set as 'low' (example: *interrupt 2 0*) [R1]. It needs to be reset each time before generating another interrupt by sending a 'high' signal level (*interrupt 2 1*). Only five external interrupt lines are can be used on the target simulator from 0 to 4.

2.5 Specification and Description Language (SDL) - ObjectGeode

The use of ObjectGeode is used in the specification phase of the software life cycle. The tool is intended to validate and execute the system specifications for a hard real-time environment. As the specifications are inserted in the SDL tool model, it is possible to perform simulations of the overall or detailed behavior of the system. For instance, the user can feed the simulator with specific values which are sent through the different objects and processes. It is also possible to trace the execution paths, enquire about the state and transition coverage, and

generate specific test cases to be recorded for later usage. All this information generated during the simulation time allows the user to check and validate the correct behavior of the system before going to the design phase.

The SDL tool is to be used in conjunction with the HRT-HOOD tool in the following way. After a certain number of objects have been specified in SDL, a translation mechanism is able to convert selected SDL objects to HRT-HOOD objects based on a set of rules (currently in preparation). Based on the main reference SDL file representing the model, the selected SDL objects are to be extracted directly in Ada code. More details of the communication mechanisms and the translation rules between the two environment are described in [R6].

2.6 ESA Software Tools

A few tools coming directly from ESTEC are used in the software development process flow. Their usage is mainly oriented for additional support of the already commercial tools described previously in this report.

The **S**PARC **I**nstruction **S**imulator (SIS) is a tool used to execute and debug up to a certain extend an executable file generated through the Alsys compiler, binder, and linker. For instance, SIS allows the user to set breakpoints and also trace through instructions at a time. It can be used to check the initial behavior of a model before using the more powerful and user friendly target simulator from Spacebel.

The other set of tools are used to facilitate the various transitions between the different tools in the software chain process. There consist of either unix scripts or *gawk* programs. The *gawk* programs are used to translate and reformat files in order to have these files as a correct format for input to the other tools. Note that these last tools are not under configuration management.

3. Programming Languages

3.1 The Ada Language

The Ada language is the main programming language used throughout this entire software development process for on-board space applications. A number of Ada language features, especially those oriented towards hard real-time applications, are used in the code, and are described in the following paragraphs.

The package *Real_Time* is the most important package related to the handling of time. It is used throughout all the different objects dealing with the interface to the real time clock and for the purposes of processing accurate delays [R5].

The package *Interrupt_Manager* serves the user interface for interrupt management [R5]. This package is used in conjunction with the HRT-HOOD object of type interrupt sporadic to define the interrupt request identification. The extracted code of the interrupt sporadic contains also the *Attach_Handler* procedure which points to the requested interrupt ID.

Example (Attached Interrupt Handler)

Attached handler for an interrupt sporadic:

INT Sporadic RTATT:

```
...
package Interrupt_Sporadic_RTATT is
  ...
  INITIAL_CEILING :   constant PRIORITY := 49;
  ...
end Interrupt_Sporadic_RTATT;
```

INT Sporadic specification:

```
...
Interrupt_Sporadic_Id : constant  Interrupt_ID := 10;
...
```

INT Sporadic body:

```
...
procedure Handle_Interrupt_Request is
   begin
      OBCS.Start;
end Handle_Interrupt_Request;
...
Attach_Handler (Handler => Handle_Interrupt_Request'Address,
                     Interrupt => Interrupt_Sporadic_ID,
                     Priority => Interrupt_Sporadic_RTATT.INITIAL_CEILING,
                     Number_of_Buffers => 1,
                     Max_Param_Area_Size => 0);
...
```

There are only four traps which can be mapped to a user defined interrupt (02, 03, 10, 11) [R5] - the trap being *Interrupt_Sporadic_ID* in the above example. The mapping between the traps and the interrupt lines of the target simulator is the following: 02 to 0, 03 to 1, 10 to 2, and 11 to 3.

A package generating interrupt - the *GENERATOR* package developed at ESTEC - based on a fixed period, can also be used for interrupt sporadic objects. Therefore, the inclusion of that package in the main Ada unit does not require the user to use the target simulator from Spacebel to generate interrupt arrivals.

Example (Programmatic Generation of Interrupt)

Ada code which forces the generation of an interrupt:

```
...
procedure Force is
   Force_Interrupt : constant MEC.INTEGER_32 := MEC."**" ( 2, Interrupt_Level );
   Interrupt_Force_Register : constant MEC.REGISTERS := MEC.IT_FORCE;
   IFR_Value : MEC.INTEGER_32;
 begin
   IFR_Value := MEC."+" ( MEC.READ ( Interrupt_Force_Register ), Force_Interrupt );
   MEC.WRITE ( Interrupt_Force_Register, IFR_Value );
end Force;

procedure thread_action is
begin
  Force;
end thread_action;
...
```

The package *Indirect_Call* provides a mechanism for calling an Ada procedure indirectly via its address [R5]. It allows to carry one parameter, which consists of a record, or none at all. The indirect addressing is also done for referencing C functions.

Example (Indirect Call)

```
...
   package Int_Indirect is new Indirect_Call.With_Param(Do_Work_Routine.Param_Rec);
begin
    Int_Indirect.One_Param (Data => Param, Code => Ops);
...
```

The use of the C language can be interfaced with Ada. This is performed by having the *pragma INTERFACE* and *INTERFACE_NAME* defined in the specification. With this declaration, the C functions are known to the Ada environment. To bind and link with interfaced C code, the bind invoke file has ro be updated. The 'interface => modules' option has to be included the path name with the .o c file (e.g. *interface => (modules => "$HOME/tool/dem32/userdem32.o $HOME/c/hello_world.o"));*)

Example (Ada-C Interface)

```
...
  private
    pragma INTERFACE (C, OPCS_WRITE);
    pragma INTERFACE_NAME (OPCS_WRITE, "_c_opcs_write");
...
```

The HRT-HOOD tools enables the user to define a specific pragma for C interface handling called (HDN_C_INTERFACE) which takes one parameter as the name of the C function. The OPCS of a particular entry is directly associated with its C pragma interface as defined in the above example.

The *pragma PASSIVE* is used to provide an Ada source code model for protected objects [R4]. It is placed in the specification of the task.

The *pragma PRIORITY* is used to assign a specific priority number to the thread of a cyclic and sporadic object and to the OBCS of a protected and sporadic objects. The highest value represents the most important thread or OBCS (i.e. it has the highest priority).

Example (Pragma PASSIVE and PRIORITY)

```
task OBCS is
   pragma PRIORITY(Buffer_RTATT.INITIAL_CEILING);
   pragma Passive;
   entry Write(Item : in Integer);
```

The packages *SEMAPHORES* and *RTX_INTERFACE* are no longer required in the Ada code used with the Alsys ERC32 compiler. The run-time system handles the ceiling priority mechanisms defined in the *RTX_INTERFACE* package. The *SEMAPHORE* package is removed after modifying the Ada code to handle the *signal* and *wait_start* routines without that specific package.

The Ada input/output (I/O) handling and processing through the *TEXT_IO* package should be done with extra care. To avoid potential problems, there should be only one main routine which does the I/O calls and also only one location where the data to be printed is stored. No handling of the I/O should be done through the use of a protected object.

The WCET (Worst Case Execution Time) analysis requires special Ada code constructs. The use of I/O packages as part of the Ada runtime, runtime heap and access types with the new operator, unbounded loops, the *Clock*, *To_Duration*, and *To_Time_Span* operations from the *Real_Time* package, exception handling within the critical code, are a few examples of the constraints. The complete set of rules is described in [R3].

On a broader scale, all of the Ada constructs and features related to the handling of types should be used to their full extend. Types should be bounded as much as possible to avoid potential risk of constraints errors, unexpected dying of task, or complete hanging of the system.

3.2 The C language

As mentioned previously in 3.1, the Ada language can be interfaced with C through the *pragma INTERFACE*. It should be noted that when a C function returns a parameter to an Ada procedure, it should always be returned as a pointer. Variables declared in Ada can be exported to C programs through the use of the *pragma EXPORT*. A complete example of C interface with Ada is part of the example described in Appendix A.

4. Personal Assessment

During the period on which I was involved on this particular software process for the development of on-board space applications, I was able to learn and apply a number of new tools, language features, and methods. The learning process started with HRT-HOOD, then SDL, and after a few months the ERC32 Cross Ada Compiler System, and finally the Spacebel tools. I believe that initially, it was absolutely necessary to fully understand all the main features of the HRT-HOOD methodology along with the type of code which can be extracted before going further in the software process flow, since this object oriented approach in a real-time environment was fairly new to me. The ERC32 Cross Ada Compiler System (ACS) was not completely new to me since I had the opportunity to work on a similar technology in my previous work experience. The real new part consisted of the Worst Case Execution Time (WCET) analysis and its specific Ada code constructs and also the compile, link, and bind invoke files modifications. The use of the third set of tools - from Spacebel - proved to be a new area of analysis for me. The schedulability simulator involved some additional support in order to better understand the features, how to interpret the output results, and especially in what way to correct the schedulability errors which could occur during the simulation. As a general personal assessment of this experience, I believe that I was able to learn and put into practice many new software engineering principles, methods, and aspects relating to all the different phases of the life cycle dealing with the development of on-board space applications. All the interactions and information gained through the use of the tools are essential for the correct and complete software project development.

A number of lessons were learned during this period of working on these hard real-time tools. The different debugging techniques to be used when trying to find a problem in the code, such as "breaking" down the code in small portions, was an excellent experience. The implementation phase - Ada code -allowed me to gain more knowledge about the language, but it especially forced me to try to use the Ada features relating for instance to type handling and real-time constraints to their full extent.

Another part of my activities was to write two documents, [R6] and [R7], which were used by the industry. The first technical report was used by Intecs Sistemi to update the HoodNICE code extractor in order for the source code to be compatible for the ERC32 Cross Ada Compiler System. The second document is being used to negociate with Verilog for the implementation of the bridging concepts between SDL and HRT-HOOD. These assigments proved to be a valuable learning experience since the reports had to be very precise and at the same time understandable for the reader who may not be an expert in SDL, HRT-HOOD, or the ERC32 compiler.

All of these different tools, language aspects, and methods were built up in a demonstration system (cf. Appendix A) which was presented at ESTEC last February and at the conference DASIA'97 (Data Systems in Aerospace) in Sevilla, Spain at the end of May.

Finally, the major shortcoming or more precisely the difficulty in using these tools, is that it

takes patience and time to be able to understand all the different interactions and usage of the tools.

5. Onboard Operations Support Software (OBOSS) Activities

The actvities concerning OBOSS were accomplised during the last two and half months of my stay at ESTEC. The main goal of OBOSS is to have a reusable software component for the ESA packet utilisation standard. All the code is in Ada and was delivered to ESTEC by CRI. It contains around sixty objects for a total of about 16,000 lines of code. My main tasks were to take all the necessary objects which allowed me to go through the different steps of defining a telecommand (TC) for a particular service (ex: Device Level Commanding operations), simulating the uplink of a TC to a spacecraft, receiving the TC, and then processing the TC according to its specific operations. It was not necessary for me to use all the code provided by CRI since I went through only one chain of activities (i.e. all the operations required for the Device Level Commands). In order to be able to work through the chain of activities described above, it was necessary to modify the code to a certain extent. Everything was compiled, linked, and bound for an ERC32 environment. The Ada tasking structure defined in the CRI code was also adapted so that protected, cyclic, and spoardic objects would have the same code structure as if there were extracted from a HRT-HOOD tool. This overall actvity involved some support from CRI in order to be able to understand critical parts of the code. For instance, it was essential to know the correct and valid way of building a telecommand and the initial stages of the uplink. It is expected that in the longer run, OBOSS will be used for the development of PROBA. Therefore, the software developed will be integrated with the various tools and language features described in the previous chapters.

The initial difficulties were to be able to understand the main chain of calls of the large amount of Ada code written for OBOSS. It is not that easy to piece together more than 60 modules and thousands of lines of code especially when it was not been written by yourself. The other main difficulty originated from the fact that it was really the first time that I was exposed to "real" onboard code. Therefore, it involved some learning of the different features associated with the code.

One of the main positive aspects of working on the OBOSS code was to be able to work on meaningful code to be used onboard satellites. This experience of dealing with "real" code will be a valuable aspect for future encounters when working on code developed to run for onboard spacecraft systems. Another positive fact was that I was forced to adapt and work my way around code that had already been written by some other software engineers.

6. Conclusion

The software process described in this document goes through all the required tools of the life cycle which contribute to the development of hard real-time applications for on-board space systems. From the initial steps of the specification with SDL to the testing and running of the executable on the target simulator, the software engineer passes through various phases. Each one of them complements the other, and the process always requires an iterative feedback between the other tools.

There are a number of positive aspects for using this specific software technology. Each tool has a dedicated function but its results and outputs can be interfaced and used very easily between the other tools. There is a close interaction with all the tools. By using this software technology, larger and more complex systems for hard-real time on-board environments can

be developed. Therefore, time and money required during the development phases is reduced. In fact by having some type of validation and simulation stages during the life cycle, the quality of the system can be increased and achieve a higher standard.

On the other hand, this technology has not yet involved a project for a large software development with for instance more than two hundred HRT-HOOD objects. Another aspect is that the automatic translation mechanisms between SDL and HRT-HOOD is not available as of right now. Finally, the industry needs to be convinced that this proposed technology is solid and can be used for the development of on-board space applications.

In conclusion, the immediate and future use of this technology and its derived tools is for the PROBA (Project for On-Board Autonomy) satellite by the industry. The goal is to prove that it is perfectly feasible to develop critical software in this particular fashion, and for it to run on a satellite platform.

Appendix A - DASIA'97 Example

The demonstration system is composed of four different sub-systems - an interrupt handler, a printing system, a C interface and a main unit system. All the different types of HRT-HOOD objects are used along with the various Ada features described in 3.1. The entire process flow - as described in the report - was presented at DASIA, except for the SDL part which is under discussion with Verilog.

The complete example is not given in this appendix. An overview of specific object Ada bodies which have particular features are presented. For the complete example including the HRT-HOOD model, a tape is available at ESTEC from WSD.

Interrupt_Sporadic *object package body:*

```
package body Interrupt_Sporadic is      -- SPORADIC HRT-HOOD object implementation
   -------- FORWARD OPERATION DECLARATIONS ----------
   procedure Handle_Interrupt_Request;

   procedure OPCS_Start(OVERRUN : in BOOLEAN ; Start_TIME : in TIME);

   ----------------    OBJECT CONTROL STRUCTURE    ----------------
   --|:OBCS_CODE
   --|:END_CODE
   -- operation Start, unbuffered
   Start_BUFFER : Start_PARAMETER_SET;

   task body OBCS is
   begin
      loop
         accept Start;
            OPCS_Start(Start_BUFFER.OVERRUN, Start_BUFFER.Start_TIME);
      end loop;
   end OBCS;

   -------- OPCS OF CONSTRAINED OPERATIONS ----------------
   procedure Handle_Interrupt_Request is
   begin
      OBCS.Start;
   end Handle_Interrupt_Request;

   procedure OPCS_Start(OVERRUN : in BOOLEAN ; Start_TIME : in TIME) is
```

```
   --|:OPCS_CODE <Start>
         begin
           SW_Sporadic_INT.Obcs.Start (Real_Time.Clock, Var_1);
      --|:END_CODE
   end OPCS_Start;

begin
   Attach_Handler (Handler => Handle_Interrupt_Request'Address,
            Interrupt => Interrupt_Sporadic_ID,
            Priority => Interrupt_Sporadic_RTATT.INITIAL_CEILING,
            Number_of_Buffers => 1,
            Max_Param_Area_Size => 0);

end Interrupt_Sporadic;
```

Producer *object from the main unit (cyclic object):*

```
...
   task THREAD is
      pragma PRIORITY( Producer_RTATT.INITIAL_PRIORITY);
   end THREAD;

   task body THREAD is
      T      : Real_Time.TIME := Real_Time.CLOCK;
   begin
      if T < RTA.SYSTEM_START_UP_TIME then
         T := RTA.SYSTEM_START_UP_TIME;
      end if;
      T := T + Producer_RTATT.THREAD_OFFSET(RTA.START_MODE);
      FOREVER:
      loop
         Real_Time.delay_until(T);
            thread_action;
         T := T + Producer_RTATT.THREAD_PERIOD(RTA.CURRENT_MODE);
      end loop FOREVER;
   end THREAD;

   ----------   OPCS OF CONSTRAINED OPERATIONS   ----------

   ----------     OPCS OF UNCONSTRAINED OPERATIONS   ----------
```

```
   procedure thread_action is
   --|:OPCS_CODE <thread_action>
         begin
           Buffer.Obcs.Write (Item);
           Back_ground_Ops.Work_More (100);
           Consumer.Obcs.Start;
           Store.Obcs.Write (4);
           Item := Item + 1;
       --|:END_CODE
   end thread_action;
...
```

Consumer *object from the main unit (sporadic object):*

```
...
   Start_PARAMETERS : Start_PARAMETER_SET;
   Start_CALLED : BOOLEAN := FALSE;

   task THREAD is
      pragma PRIORITY( Consumer_RTATT.INITIAL_PRIORITY);
   end THREAD;

   task body THREAD is
      T       : Real_Time.TIME := Real_Time.CLOCK;
      Start_BUFFER    :   Start_PARAMETER_SET;
   begin
      FOREVER:
      loop
         OBCS.WAIT_Start(Start_BUFFER);
         OPCS_Start(Start_BUFFER.OVERRUN, Start_BUFFER.Start_TIME);
      end loop FOREVER;
   end THREAD;

   task body OBCS is
   begin
      loop
         select
            accept Start do
               -- store parameters
               Start_PARAMETERS := (FALSE, Real_Time.CLOCK);
```

```
                -- if not already pending prepare execution,
                -- otherwise over-write previous request.
                if not Start_CALLED then
                   Start_CALLED := TRUE;
                else
                   Start_PARAMETERS.OVERRUN := TRUE;
                end if;
             end Start;
          or
          when Start_Called =>
             accept WAIT_Start(THE_PARAMS : out Start_PARAMETER_SET) do
                THE_PARAMS := Start_PARAMETERS;
                Start_PARAMETERS.OVERRUN := FALSE;
                Start_CALLED := FALSE;
             end WAIT_Start;
          or
             terminate;
          end select;
       end loop;
    end OBCS;

    procedure OPCS_Start(OVERRUN : in BOOLEAN ; Start_TIME : in TIME) is
    --|:OPCS_CODE <Start>
         begin
          Buffer.Obcs.Read(Item);
          Store.Obcs.Write (5);
          Back_Ground_Ops.Work_More (50);
       --|:END_CODE
    end OPCS_Start;
...
```

Buffer *object from the main unit (protected object):*

```
...
    Head :  Buffersize := 0;
    Tail :  Buffersize := 0;
    Store :  array (Buffersize) of Integer:= (others => 0);
       ----------   OPERATION(S)   ----------
    function Increment(I : in Integer) return Integer;
    --------   FORWARD OPERATION DECLARATIONS   ----------
```

```
procedure OPCS_Write(Item : in Integer);
-- standard operation to contain user code
procedure OPCS_Read(Item : out Integer);
-- standard operation to contain user code

----------    OBJECT CONTROL STRUCTURE    ----------

task body OBCS is
begin
   loop
      select
         accept Write(Item : in Integer) do
            OPCS_Write(Item);
         end Write;
      or
         accept Read(Item : out Integer) do
            OPCS_Read(Item);
         end Read;
      or
         terminate;
      end select;
   end loop;
end OBCS;

----------   OPCS OF CONSTRAINED OPERATIONS   ----------

procedure OPCS_Write(Item : in Integer) is
--|:OPCS_CODE <Write in Integer>
     begin
       Store (Head) := Item;
       Head := Increment (Head);
   --|:END_CODE
end OPCS_Write;

procedure OPCS_Read(Item : out Integer) is
--|:OPCS_CODE <Read out Integer>
     begin
       Item := Store (Tail);
       Tail := Increment (Tail);
   --|:END_CODE
```

```
   end OPCS_Read;

   ----------------     OPCS OF UNCONSTRAINED OPERATIONS   ----------------
   function Increment(I : in Integer) return Integer is
  --|:OPCS_CODE <Increment in Integer Integer>
         begin
           if I < Buffersize'Last then
             return (I+1);
           else
             return 0;
           end if;
      --|:END_CODE
   end Increment;
...
```

Print_Tool *object from the printing unit (cyclic object):*

```
...
   procedure thread_action is
   --|:OPCS_CODE <thread_action>
        Transfer_Array : Store.T_Array := (others => 8);
        Counter : Integer;
          begin
            Store.Obcs.Read (Transfer_Array, Counter);
            if (Counter > Low) then
              Counter := Counter - 1;
              for I in Low .. Counter loop
                 Prints.Output (Transfer_Array(I));
              end loop;
              Prints.Output (9);
            end if;
      --|:END_CODE
   end thread_action;
...
```

Buffer_IF *separate procedure from the C interface unit (protected object):*

```
separate (Buffer_IF)
```

```
procedure OPCS_Write( Item : in Integer) is
   --------    DESCRIPTION --------
--|:OPCS_CODE <Write in Integer>
      begin
        C_OPCS_Write (Item);
   --|:END_CODE
end OPCS_Write;
```