Stellingen
behorende bij het proefschrift:

DEVELOPMENT OF
ON-BOARD EMBEDDED REAL-TIME SYSTEMS:
AN ENGINEERING APPROACH

Tullio Vardanega


2. De notie architectuur houdt in: (i) talen om systemen te beschrijven; en (ii) een aantal concepten, ontwerpregels, ontwerpprincipes en richtlijnen voor verifieerbare transformaties van beschrijvingen in de ene taal naar een andere taal.


Deze definitie heeft twee onverwachte implicaties: (i) een implementatie is een recursieve transformatie met begrensde diepte; (ii) een benadering vloeit voort uit vervorming (door toevoeging aan of aftrekking van) in een transformatiestap. Technologie voor algemene doeleinden en COTS componenten bevorderen het produceren van benaderingen meer dan dat van implementaties.

3. Het belang van op objecten georiënteerde methoden (OOM), in tegenstelling tot oudere gestructureerde methoden, ligt in de gelijkheid van notatie door OOM gebruikt voor analyse en ontwerp. Dit vermindert de complexiteit van de overgang tussen die twee fasen.


Zo'n vermindering is voordelig en zou op andere segmenten van het software ontwikkelproces toegepast moeten worden. De notie van design framework in hoofdstuk 3 van dit proefschrift geïntroduceerd doet dit door vermindering van de complexiteit van de overgang tussen ontwerp en implementatie.

4. Alexanders' patroon is een regel die beschrijft wat men moet doen om de door dat patroon gedefinieerde entiteit te genereren.


Het concept van Computational Model in hoofdstuk 3 van dit proefschrift geïntroduceerd mag beschouwd worden als een coherente verzameling van patronen voor de constructie van real-time systemen.

5. Onduidelijkheid in gebruiker- en interface-vereisten en een verscheidenheid van timing-vereisten komen vaak voor bij de ontwikkeling van real-time boordsystemen.

De watervalmethode is te star om zich aan deze moeilijkheden aan te passen. Een ontwikkelproces is dus nodig dat zo wel incrementele ontwikkeling als overlap en terugkoppeling tussen fasen kan steunen. De prototyping methode is te geconcentreerd op implementatie. De Boehm spiraal methode is zo gecompliceerd dat deze minder efficient is dan de in hoofdstuk 4 van dit proefschrift gepresenteerde methode.

6. Uit sociologische perspectief, is een internationaal centrum als ESTEC een zeer opmerkelijk experiment in de creatie van het 'Europese type'. Maar het experiment leidt aan enkele serieuze gebreken, het verspilt te veel kracht en, dus, is zijn menselijke opbrengst veel minder dan zijn invoer.

7. Niet-precieze berekeningen zijn een aantrekkelijke middel om een verhoogd en effectiever gebruik van real-time toepassingen te krijgen.


Ten koste van wat run-time overhead, kunnen niet-precieze berekengingen door Ada 95's time-triggered ATC en dynamisch prioriteiten uitgevoerd worden.


Deze oplossing laat de toevoeging van een potentiële onbegrensd autonome bestuursfunctie van boordsystemen toe, zoals in dit proefschrift beschreven wordt.

8. Ruim 80% van de kosten van software voor conventionele systemen wordt globaal toegeschreven aan onderhoud na het uitbrengen, waarvan de helft typisch naar correctief onderhoud gaat.


Het gebruik van geautomatiseerde code generatie als een middel naar het "goedkoper, sneller, beter" paradigma toe, moet gebaseerd zijn op een bewustzijn van dit fenomeen. Op zichzelf, kan het wel een middel naar het "sneller" toe zijn, maar het kan zonder goed software engineering toch niet naar het "goedkoper" en "beter" toe.


Deze eisen overlappen met die die aan conventionele boordsystemen gesteld worden. Dus, de ontwerp benadering beschreven in dit proefschrift is een kant-en-klare oplossing voor de ontwikkeling van elektronische producten voor de consument.

10. Het voorrecht van de oudere is de herinnering aan de verleden tijd. Het voorrecht van de jeugd is de aanbouw van de toekomst. Vooruitgang zonder verlies van bestaande waarden heeft eerder voor allebei.

Tullio Vardanega (Delft, 1998)
Development of
On-Board Embedded Real-Time Systems:
An Engineering Approach

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,

op dinsdag 6 oktober 1998 te 10:30 uur

door

TULLIO PIETRO MARIA VARDANEGA
Scienze dell’Informazione
Università degli Studi di Pisa (Italië)
geboren te La Spezia (Italië).
Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. J. van Katwijk

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof.dr.ir. J. van Katwijk, Technische Universiteit Delft, promotor
Prof.dr. A. Burns, University of York, Verenigde Koningrijk
Prof.dr. H. Koppelaar, Technische Universiteit Delft
Prof.dr. F. Panzieri, Università di Bologna, Italië
Prof.dr. J.-A. de la Puente, Universidad Politecnica de Madrid, Spanje
Prof.dr. K. De Vlaminck, Katholieke Universiteit Leuven, België
Prof.dr. J. Zalewski, University of Central Florida, Verenigde Staten

© T.P.M. Vardanega, 1998
Al ken ik alle geheimen en alle wetenschap:
als ik de liefde niet heb,
ben ik niets.

1 Korintiërs 13.2.
# Contents

1 Problem Definition .............................................. 1
   1.1 Introduction ............................................. 1
      1.1.1 Characterisation of the Application Domain .......... 1
      1.1.2 The Rise and Stride of On-Board Software .......... 3
   1.2 Positioning of the Work .................................. 4
   1.3 Discussion of Emerging Requirements ...................... 5
      1.3.1 Process Requirements ................................ 6
      1.3.2 Process and Product Requirements .................... 6
      1.3.3 Product Requirements ................................ 8
   1.4 Research Objectives ....................................... 9
   1.5 Structure and Outline of the Work ......................... 10

2 Characterisation of On-Board Embedded Systems ............... 13
   2.1 Introduction ............................................. 13
   2.2 Overview of Typical Platform Functions .................. 14
      2.2.1 Telecommands ....................................... 15
      2.2.2 Telemetries ......................................... 17
      2.2.3 Data Handling Control ............................... 17
      2.2.4 Attitude Determination and Orbit Control .......... 18
   2.3 The Challenge of Integration .............................. 20
   2.4 Summary of Important Characteristics ..................... 22

3 Weaknesses of and Amendments to the Current Software Process 25
   3.1 Introduction ............................................. 25
   3.2 The Present Status ....................................... 26
      3.2.1 Software Process ................................... 26
      3.2.2 Support Technology .................................. 28
   3.3 Cheaper, Faster, Better ................................... 31
      3.3.1 Deficiencies of the Software Requirements Definition Phase 33
      3.3.2 Deficiencies of the Design Definition Phase and Technology 35
   3.4 Solution Space ........................................... 36
## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1 Evolutionary versus Revolutionary Amendments</td>
<td>36</td>
</tr>
<tr>
<td>3.4.2 Scope of the Proposed Amendments</td>
<td>38</td>
</tr>
<tr>
<td>3.4.3 Proposed Amendments to the Current Software Process</td>
<td>39</td>
</tr>
<tr>
<td>3.4.4 Conceptual Definition of Computational Model</td>
<td>41</td>
</tr>
<tr>
<td>3.4.5 Practical Instances of Computational Model for On-Board Systems</td>
<td>42</td>
</tr>
<tr>
<td>3.4.6 Our Choice</td>
<td>46</td>
</tr>
<tr>
<td>3.5 Relation to Other Practices</td>
<td>47</td>
</tr>
<tr>
<td>3.5.1 Process Aspects</td>
<td>47</td>
</tr>
<tr>
<td>3.5.2 Technology Aspects</td>
<td>48</td>
</tr>
<tr>
<td>3.6 Summary of Discussion</td>
<td>49</td>
</tr>
<tr>
<td><strong>4</strong> An Evolutionary Approach to the Construction of New-Generation Systems</td>
<td>51</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>51</td>
</tr>
<tr>
<td>4.2 Methodological Approach</td>
<td>52</td>
</tr>
<tr>
<td>4.2.1 Outline of the Proposed Approach</td>
<td>52</td>
</tr>
<tr>
<td>4.2.2 Relation to Other Development Approaches</td>
<td>54</td>
</tr>
<tr>
<td>4.2.3 Rationale of the Proposed Approach</td>
<td>55</td>
</tr>
<tr>
<td>4.3 Supporting Concepts and Technology Requirements</td>
<td>58</td>
</tr>
<tr>
<td>4.3.1 Essential Elements of the Proposed Solution</td>
<td>58</td>
</tr>
<tr>
<td>4.3.2 Technology Requirements</td>
<td>60</td>
</tr>
<tr>
<td>4.4 HRT-HOOD as the Centre of the Development Process</td>
<td>66</td>
</tr>
<tr>
<td>4.5 Enabling Technology</td>
<td>70</td>
</tr>
<tr>
<td>4.5.1 The HRT-HOOD Computational Model</td>
<td>70</td>
</tr>
<tr>
<td>4.5.2 Ada Implementation</td>
<td>73</td>
</tr>
<tr>
<td>4.5.3 Static Analysis</td>
<td>75</td>
</tr>
<tr>
<td>4.6 Analysis of the Proposed Solution</td>
<td>85</td>
</tr>
<tr>
<td><strong>5</strong> Illustrative Example</td>
<td>89</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>89</td>
</tr>
<tr>
<td>5.2 Construction and Analysis of an HRT-HOOD System</td>
<td>89</td>
</tr>
<tr>
<td>5.3 Step B.1: Real-Time Design</td>
<td>90</td>
</tr>
<tr>
<td>5.4 Step B.2: Binding to Computational Model</td>
<td>95</td>
</tr>
<tr>
<td>5.5 Step B.3(a): Extraction of Design Requirements</td>
<td>98</td>
</tr>
<tr>
<td>5.6 Step B.3(b): Generation of WCET Profile</td>
<td>101</td>
</tr>
<tr>
<td>5.7 Step B.4: Feedback to Design</td>
<td>105</td>
</tr>
<tr>
<td>5.8 Summary</td>
<td>110</td>
</tr>
<tr>
<td><strong>6</strong> Proof of Concept</td>
<td>113</td>
</tr>
<tr>
<td>6.1 Historical Note</td>
<td>113</td>
</tr>
<tr>
<td>6.2 Concept Establishment</td>
<td>113</td>
</tr>
<tr>
<td>6.3 Approach to Assessment</td>
<td>115</td>
</tr>
<tr>
<td>6.4 Definition Phase: The Olympus Case Study</td>
<td>116</td>
</tr>
<tr>
<td>6.4.1 Positioning of the Project</td>
<td>116</td>
</tr>
</tbody>
</table>
6.4.2 System Architecture ................................................. 117
6.4.3 Implementation Experience ...................................... 118
6.4.4 Discussion .......................................................... 120
6.5 Consolidation Phase: The Eureka Case Study .................. 122
   6.5.1 Positioning of the Project ................................. 122
   6.5.2 System Architecture ........................................ 123
   6.5.3 Implementation Experience .............................. 125
   6.5.4 Discussion .................................................... 128
6.6 Maturity Phase: The ERA Project .................................. 129
   6.6.1 Positioning of the Project ................................ 129
   6.6.2 System Architecture ....................................... 129
   6.6.3 Highlights of Early Implementation Experience ........ 131
   6.6.4 Discussion .................................................... 132
6.7 Summary ............................................................. 132

7 Concluding Remarks .................................................. 135
   7.1 Recalling the Thesis Objective ................................. 135
   7.2 Reviewing the Accomplishments .............................. 136
   7.3 Future Perspective .............................................. 138
      7.3.1 Relevance to Other Application Domains .............. 138
      7.3.2 Future Work in the Space Domain ..................... 140

A Glossary ............................................................... 143

B Summary ............................................................... 147
   B.1 Subject .......................................................... 147
   B.2 Problem .......................................................... 148
   B.3 Solution .......................................................... 148

C Samenvatting .......................................................... 151
   C.1 Onderwerp ........................................................ 151
   C.2 Probleem ......................................................... 152
   C.3 Oplossing ......................................................... 152

D Curriculum Vitae ..................................................... 155

Bibliography ............................................................. 156
List of Figures

1.1 Growth of Memory Budget in European Satellite Systems. ..................... 7

2.1 Satellite System Building Blocks. .................................................. 14
2.2 Typical Organisation of On-Board Processing Activities. ...................... 19
2.3 Principle Architecture of Ideal ICDS System. .................................... 21

3.1 The V-Model Development Cycle. .................................................. 27

4.1 The Proposed Software Process. ...................................................... 53
4.2 Design Activities in the Proposed Process. ....................................... 59
4.3 Feedback-Based Iterative Design Flow. .......................................... 64
4.4 Design Activities in Operation. ...................................................... 69
4.5 Task Templates. .............................................................................. 72
4.6 The Proposed Toolset. ...................................................................... 86
4.7 A Schematic View of the Proposed Approach. ................................... 87

5.1 A Simple HRT-HOOD System. .......................................................... 92
5.2 HRT-HOOD Object Description Skeleton Fragment (Producer). ............... 94
5.3 "With" Preamble of Producer (body). .............................................. 96
5.4 Producer Real-Time Attributes (Fragment). ....................................... 96
5.5 Operation of Producer (separate body). .......................................... 97
5.6 Producer (body). ............................................................................ 98
5.7 Consumer (spec). ........................................................................... 99
5.8 Consumer OBCS (body). ............................................................... 100
5.9 Operation of Consumer (separate body). ........................................ 101
5.10 Consumer (body). ................................................................. 102
5.11 Buffer (spec). .............................................................................. 103
5.12 Application User Configuration File (Fragment). .............................. 104
5.13 Application Execution Skeleton File (Fragment). .............................. 105
5.14 Scheduler Simulator Output (Gantt). ............................................. 110

6.1 Chronology of Development. .......................................................... 114
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>The Olympus AOC Hardware Architecture.</td>
<td>117</td>
</tr>
<tr>
<td>6.3</td>
<td>The Eureca DHC Hardware Architecture.</td>
<td>124</td>
</tr>
<tr>
<td>6.4</td>
<td>Worst-Case Software Schedule.</td>
<td>125</td>
</tr>
<tr>
<td>6.5</td>
<td>Scheduling Analysis Reports.</td>
<td>126</td>
</tr>
<tr>
<td>6.6</td>
<td>The ERA Control System Architecture.</td>
<td>130</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Overview of Driving Requirements. ........................................ 57
4.2 Disallowed "Implemented By" Relationships. .......................... 68
4.3 Allowed & Disallowed "Use" Relationships. ............................ 68
4.4 Timing Characteristics of Basic Run-Time System Primitives (μs). 75
4.5 Types of PO Calls. .................................................... 78
4.6 PO Call Overhead. .................................................... 79
4.7 Call Overheads Accountable for Blocking. ............................ 80
4.8 Task Administration Overhead Bounds. ............................... 81

5.1 Scheduling Analysis Report (Fragment). ............................. 107
5.2 Fragment of Scheduler Simulator Output (Statistics). ................. 108
5.3 Fragment of Scheduler Simulator Output (Schedule). ................ 109

6.1 Areas of Concern. ..................................................... 115
6.2 Expectation Profile (Olympus Case). .................................. 116
6.3 Case Study vs Original Software (Development Viewpoint). ........ 118
6.4 Case Study vs Original Software (Performance Viewpoint). ........ 119
6.5 Porting of Case Study on 1750A Processor. .......................... 120
6.6 Expectation Profile (Eureca Case). ................................... 123
6.7 Original Software vs Redevelopment. .................................. 127
6.8 Expectation Profile (ERA Case). .................................... 130
Chapter 1

Problem Definition

1.1 Introduction

This thesis presents an innovative engineering approach to the development of the real-time software embedded on board new-generation satellite systems. The work presented in this document was performed by the author in his capacity of staff member of the Research & Technology Centre of the European Space Agency based at Noordwijk, The Netherlands.

The main thrust of the work originated from the author’s ambition to address the issues arising from the development of new-generation on-board embedded real-time systems by a research project with high-profile academic value and equally strong emphasis on the direct applicability of the results to the industrial practice in the domain.

1.1.1 Characterisation of the Application Domain

Space applications can be broadly divided in the two classes of manned and unmanned applications. Although expectedly expanding in the medium-term future, the former class presently represent a comparatively tiny, high-cost minority. The latter, instead, constitute the largest, most lucrative, steadily growing proportion of current space applications. And it is on selected components of the latter which this thesis shall specifically focus.

In spite of their relying upon a common core of navigation and communication functions, manned and unmanned systems vastly differ in the nature of all other functionality. The presence (or absence) of humans connected with the operation of the spacecraft is, in fact, the crucial driver to the architecture, the operation and the technology required for the system.

Albeit the design of manned space systems presents a number of extreme technological demands, unmanned space systems exhibit the most distinguishing characteristic of operating remotely from the access of humans for the entirety of their life time. Whereas the unprecedented (and successfully repeated) case of the in-orbit repair of the Hubble telescope configures as a
unique exception, in fact, the typical life cycle of satellite systems lends itself to only limited forms of maintenance. That aspect colours the design and implementation of such systems of an engrossing technical challenge.

Satellite systems have a relatively long operational life-time (typically inferior 10 years) and an equally long development cycle (typically about 3-4 years). Consequently, the technological obsolescence which incurs between two successive spacecraft generations makes them inherently closer to the one-off product concept than to the more business-like notion of product family. Hence, the vast majority of space systems in this category are best described as one-off, long-lived, remotely-operated, limited-access systems, highly impervious to conventional forms of corrective, adaptive and perfective maintenance.

Satellite systems operate in a harsh and hostile environment. This demands the use of hardware components designed and produced so as to be specially resilient to radiation, heavy particles and other space effects [Benedetto, 1998]. On top of this, the system must be capable of operating with low to very low power consumption profiles so as to maximise the duration of its operational life.

For these reasons, the production and the qualification of the hardware components for use on board satellite systems are significantly more complex than the equivalent process for conventional ground technology. The low relative volume of the space business further adds to that in driving the price of space qualified components to very high unitary costs.

As a result of the high cost and low demand, the most part of the required space qualified hardware technology does not develop as rapidly as conventional technology for ground-based applications. In fact, the technological gap between the two domains typically situates in the order of 8-10 years.

Albeit slow-paced, however, the miniaturisation of solid-state electronics, sensors, actuators, optics and many other space technologies continues unabated. The availability of ever-smaller satellite components has lately become an important enabling factor for the (re-)appearance of the so-called Small-Satellite Missions. This phenomenon opposes to the customary trend to ever-larger-sized and more sophisticated satellites and has vast consequences on the process employed to implement and operate satellite missions. The impact of this phenomenon and the lessons learned from recent experience of application are lucidly analysed by [Van der Ha, 1995] from the perspective of the mission and operations management.

The concept of small-satellite mission is only the most conspicuous indicator of the increasing interest manifested by Space Agencies and commercial builders in the "cheaper, faster, better" paradigm. (The feasibility of which has already been demonstrated for several years by Universities and amateur radio organisations.) In essence, the "cheaper, faster, better" paradigm denotes the urge to attain increased operation capability, decreased power consumption, lower launch mass and weight and shorter time to market within a reduced cost envelope. Smaller satellites are obviously better equipped to meet these goals, which explains their present flourishing renaissance.

When applied to the design and implementation of suitable on-board processing units, the emerging paradigm also compels satellite builders to achieve increased utilisation of ground technology [ESA, 1995b]. This particular trend, however, configures as a long-term process, of which
we are presently seeing only the initial steps.

1.1.2 The Rise and Stride of On-Board Software

The early satellite systems almost exclusively employed hardwired components and were consequently based on highly inflexible custom architectures. With time, however, the operation complexity of the system has grown to the point that the use of programmable processing components has become a necessary measure.

This has progressively led to the introduction and utilisation of microcomputers in the control and operation of several on-board functions. With low-power consumption being the main limiting factor, though, the available technology for the construction of computer-based on-board control units has borne its share of constraints and limitations. These include, for example:

- low-throughput microprocessors (typically up to 1.25 MIPS 16-bit technology)
- very limited cacheless primary memory space (typically less than 128k 16-bit words)
- no conventional secondary mass memory support other than in the form of high-capacity solid-state recorders employed as temporary FIFO-like storage for bursts of raw data produced from payload equipment and destined to ground processing.

The first noticeable utilisation of software on board satellite systems dates back to the early 70s. At that time, software was solely used in the implementation of a restricted set of control functions typically limited to the on-board handling of ground commands. Software has since established itself as an increasingly important component in the on-board operation of satellite systems.

Nowadays, with the emergence of such driving requirements as shorter time to market, increased responsiveness and maximised mission product, software pervades an ever-growing proportion of the on-board functions. Accordingly, new-generation satellite systems tend to become increasingly software-intensive.

While software has rapidly turned central to the construction, operation and performance of satellite systems, this phenomenon has been responded by a far slower advance towards the use of more modern development concepts and supporting tools.

As a result of that, on-board software is still typically developed on comparatively outdated and poorly supportive technology, deployed on tightly resource-constrained computer systems and cast within rigid and inflexible software architectures.

This frustrating condition primarily originates from the special economic and industrial characteristics of the space sector introduced earlier. As far as the software technology is concerned, the most determining factors can be briefly summarised as follows:

- owing to the hardware limitations inherent to space systems, the software technology suitable for use on board satellite systems is, in many respects, non-conventional (in particular, for its being embedded in severely resource-constrained on-board computers) and, consequently, often unable to fully exploit many well established but resource-eager concepts, practices and techniques normally employed in conventional computing systems;
• furthermore, despite contributing to the most part of the satellite’s operation, on-board software still accounts for a comparatively modest proportion of the overall cost to completion of the space system (about 20% in Agency space projects in 1994, according to [ESA, 1996]). Not surprisingly, therefore, the combination of the extremely low volume of the European space software business (which [ESA, 1996] estimates to a mere 1% of the total volume of the overall software business in Europe in 1994) and the comparatively small development costs of the on-board software component of space systems tend to yield modest short-term interest in the adoption of more advanced technology coupled with the (understandable) reluctance of the technology providers to accommodate the specific needs of space users;

• finally, in order to fit within the hardware-dominated concept of the space system, the software implementation of the designated on-board functions is still typically expected to provide the same assurance of deterministic behaviour as provided by the hardware; hence, software architects are compelled to keep the system simple and rigid atop of any other consideration.

The above limiting factors were initially met by the actual simplicity of the early software systems. That also determined the initial preference for the use of rigid and simple-minded software architectures which still continues today.

On the whole, spacecraft builders have the understandable tendency to stick as long as possible with admittedly conservative, yet well-known, engineering solutions. This approach, in fact, is perceived to yield a lower risk than the one arising from any transition to more advanced – but novel and unproven – technology. We contend, however, that the requirements arising from the "cheaper, faster, better" paradigm will challenge the conservatism in the architecture of the on-board software product and the organisation of the relevant development process.

1.2 Positioning of the Work

This thesis is situated in the context described in section 1.1 and aims to accomplish the following general objectives:

• to discuss the deficiencies of the current practice for the development of on-board software in the face of the emerging requirements; and

• to propose specific enhancements to the process and associated enabling technology which help industry meet those requirements and can be practically and cost-effectively implemented by space industry in the short-term period.

The non-disruptive introduction of amendments to an established software development process is not an easy endeavour. The mere consideration of the relative cost and weight of the on-board software component with respect to the overall satellite mission cost, in fact, suggests that the objective to solely amend the current software process and technology can hardly be a
driving factor. Other, more immediately crucial areas of the satellite technologies exist, in fact, whose needs are ponderous enough to capture the attention of the developer.

Amongst many other components, the on-board processor and the associated hardware technology have definitely entered the group of critical technologies. Hence, it is apparently more practical that the introduction of software advances be promoted in conjunction with the occurrence of important transitions in the reference processor technology.

Interestingly in that respect, the early 90s have seen the start of the transition towards more advanced processor technology (yet, still with no cache and no memory hierarchy) than the 16-bit technology presently in use. The designated target of such a transition, in Europe, is the 32-bit SPARC-based computing core (ERC32 for short) recently developed under contract 9848/92 [ESA, 1992a, Saab Ericsson Space, 1997] funded by the European Space Agency as part of its research and technology programme. The earliest hardware components of the ERC32 technology have become available for flight utilisation since the last quarter of 1996 and are presently included in the system baseline of a number of on-going projects funded by the European Space Agency.

Whereas it appears that all new space developments will consider using the novel 32-bit technology, it is not equally clear how rapid the actual transition from the old technology will eventually be. Any such process, in fact, carries important consequences for industrial developers in terms of the complexity, time demand and inherent risk associated with the implementation and qualification of all the required modifications to the routine industrial production chain.

On top of this, the embedded nature of on-board systems causes the move from one processor hardware to another to also typically determine changes to the associated software factory. Hence, there is likely to be a sufficiently large window of opportunity for suitable amendments to the current software technology to win the attention of the industry segment concerned.

The need to reconsider the current software process as part of the overall technological change is distinctly asserted by the emergence of the requirements discussed in the following.

1.3 Discussion of Emerging Requirements

Most of the programmatic directives (e.g.: [ESA, 1997, ESA, 1995b, ESA, 1996]) mission specifications (e.g.: [Teston et al., 1997]) and Invitations to Tender recently issued by the European Space Agency for future technology projects abide by the strong emergence of the "cheaper, faster, better" paradigm.

As also observed by [Van der Ha, 1995], full compliance with the emerging paradigm calls for important amendments to the current development process and considerable enhancements to the resulting product.

In the following, we shall discuss a number of distinctive requirements which originate from the "cheaper, faster, better" paradigm and impact the software development process and the software product for use on board new-generation satellite systems.

For each selected requirement we: (1) specify the intended target of the requirement, that is the process and / or the product; (2) present the main rationale for the requirement; and (3) outline
the anticipated technical consequences from the requirement which are of most interest for the scope of our work.

1.3.1 Process Requirements

R.1 SHORTER TIME TO MARKET

Rationale: The satellite market, though small, (e.g.: telecommunication and broadcasting, earth observation, weather forecasting) is opening to global competition; this will result in much sharper customer demands, especially in the respect of the required delivery time; this will, in turn, compel developers to allow for considerably compressed development schedule, with the consequent shortening of all the development activities in the process.

Technological Consequence: Development time (and, linearly increasing with it, development cost) is presently dominated by testing, which typically absorbs up to 60% of the overall development effort. Such a distribution of effort is clearly ill-balanced and fragile in the face of the demand to compress the duration and cost of the development. This is especially the case as the most part of the typical test activities is inherently centralised and sequential, for it consists of hardware-based dynamic testing, the base duration of which is not easily shortened without detracting from the ultimate quality of the end product.

Requirement R.1 reflects the fact that economic and commercial drivers urge the development schedules to get significantly shorter. Development schedules at satellite level are in fact being confronted with the urge to shorten from the typical 3-4 years to about 18-24 months [ESA, 1997, ESA, 1995b, Van der Ha, 1995].

This will have a dramatic impact on the entire development process. In particular, in view of the trend to increasingly more software-intensive systems, it shall especially challenge the quality and productivity of the software process.

1.3.2 Process and Product Requirements

R.2 INTEGRATION OF CONTROL ACTIVITIES

Rationale: Development, qualification and integration costs may be significantly lowered by the reduction of hardware and harness requirements; this option calls for the move towards a more aggressive integration of the software-implemented control activities on a single computer, in the place of the federation of computers currently used to host control and service functions. This transition is now technologically possible as more powerful processors, such as those based on the ERC32 computing core, are becoming available.

Technological Consequence: Satisfaction of the demand for integration brings together control activities with fairly diverse timing and processing requirements. This, in turn,
makes software integration a most critical stage of the satellite integration process. This, amongst other things, contributes to making the use of simple-minded fixed cyclic scheduling less convenient and practicable.

R.3 INCREASED FLEXIBILITY OF IN-FLIGHT MAINTENANCE

**Rationale**: Not all of the operation activities to be performed on board in the course of the mission can always be frozen and embedded in the on-board system at the pre-launch assembly of the satellite. In fact, the mission product of a satellite mission is higher the easier the means for in-flight corrective, adaptive and perfective maintenance of the operation sequences. Interestingly, this requirement is also captured by [Van der Ha, 1995] under the notion of "design for operability".

**Technological Consequence**: The need to allow for more comfortable upload and execution of new operation sequences is obviously better served by the separation of micro- (kernel-level) from macro- (operation-level) scheduling so that the latter can be upgraded without greatly impacting, or being impacted by, the execution behaviour of the former. This separation is difficult to achieve if the application is designed around a fixed cyclic scheduling scheme.

Requirement R.2 basically anticipates that on-board processing units will increasingly become *software-intensive*. In fact, the amount of memory (ROM and RAM) required for future systems is presently predicted to increase from the current figure of 128-256 kilo-bytes (equivalent to 15-25,000 source lines on 16-bit technology) to the 0.8-1.2 mega-bytes of new projects (equivalent to 50-60,000 source lines on 32-bit technology) in the next few years. Figure 1.1 graphically depicts the impressive trend observed in this respect in European space projects over the last two decades. This demand challenges industry to the development of an increasingly more complex software product within a significantly more productive process.

![Figure 1.1: Growth of Memory Budget in European Satellite Systems.](image)

---

*Development of On-Board Embedded Real-Time Systems: An Engineering Approach*
The software productivity rate typically observed across recent developments funded by the European Space Agency presently lies in the range 0.75-1.0 delivered source lines per hour for on-board systems comprised of 15-25,000 source lines. This shall at least quadruple (on account of twice as big software to be delivered in half the time) for new systems to fit the emerging paradigm. This objective will not be achievable without rectifying the present ill-balanced distribution of effort across the development phases and significantly improving the productivity of each phase.

Requirement R.3 addresses a part of the process (the operation and maintenance phase) which falls outside the scope of this thesis. The requirement, however, is presented here as it may effect the choice of the architecture of the software products for new-generation systems and may, therefore, be used to challenge the suitability of the overall engineering approach put forward by this work.

1.3.3 Product Requirements

R.4 INCREASED RESPONSIVENESS

**Rationale** : Scientific and remote-space missions increasingly call for the ability to capture and react *autonomously* (i.e. independently of direct intervention from ground) on the occurrence of mission-level and system-level events. The occurrence of such events may be difficult to accurately predict and schedule ahead of time in the initial operational profile of the mission.

**Technological Consequence** : Event polling, which is the most commonly used technique, is exceedingly resource-intensive and exposed to latency of response. Such latency reflects the rigidly static nature of the relevant service schedule, which can hardly be established with a sufficient degree of accuracy in the absence of detailed knowledge about the actual operation scenario. (This is particularly critical, for instance, in scientific missions or during contingency operations.)

R.5 GRACEFUL AND COMMANDABLE DEGRADATION

**Rationale** : Satellite control systems operate remotely from their ground control centers and are increasingly required to operate *autonomously* from ground control. Increased on-board autonomy, in fact, translates into reduced need for ground coverage, which is an important factor to the reduction of the mission cost. Hence, the on-board system needs to ensure *localised*, *predictable* and *programmable* degradation of service under contingency situations (e.g.: transient overload due to multiple sporadic events demanding service).

**Technological Consequence** : Fixed execution schedules are particularly exposed to the effects from failures in the time domain (e.g.: overrun) and generally unable to support the confinement of anomalies. Hence, they do no fit well in the anticipated scenario.
R.6 MAXIMUM UTILISATION OF AVAILABLE PROCESSING POWER

Rationale: Space-qualified processing power is costly. In fact, the present cost of 1 MIPS on space-qualified technology lies well over 100 times higher than the equivalent on conventional technology. This high cost can only be tolerated against the guarantee of maximised mission product.

Technological Consequence: Static cyclic scheduling is highly inefficient in this respect for it is inherently exposed to significant under-utilisation of available processing resources. The actual CPU utilisation observed for a typical on-board control application running on a 16-bit 1-MIPS processor ranges between 25% (mean) and 60% (real worst-case) while being frequently dimensioned to 80% of the available CPU time on the ground of greater safety. The potential under-utilisation of the already limited processing resources which ensues from that architectural choice may thus be important. The artificial limits imposed on the allowable processing load also frustrate the urge to add flexibility to the operation profile of the satellite (cf. requirement R.3). This would be especially inconvenient in the face of the anticipated transition towards the more powerful 32-bit processor technology. The very point of that transition would in fact be to facilitate the introduction of additional processing-intensive functions in the on-board system. The accomplishment of this objective, however, calls for the provision of flexible and yet predictable scheduling schemes to accommodate all the diverse processing requirements.

Requirements R.4-R.6 suggest that the on-board control activities to be implemented in software are likely to become increasingly concurrent (on account of the demand to accommodate more parallel control and service activities) and time-critical (on account of the demand for more reactivity and predictability). These requirements collectively demand that more attention be paid to the real-time nature of the on-board control software in a fashion, though, which achieves the required verification without adding excess cost and complexity to the testing phase. This implies a major change to the current development paradigm, which has the present test activities dominated by dynamic testing and can hardly bear important increases in the real-time complexity of the system under test.

1.4 Research Objectives

This thesis addresses the development of software embedded in the control computers on board new-generation satellite systems. Emphasis in this work is placed on the process and product enhancements required to accommodate the increasing real-time complexity of those systems in the changing development scenario.

The earlier discussion has highlighted the main problem areas which we especially want to address in our work, and the solution space within which we have to operate.

The key problem areas distinctly emerge from the discussion in section 1.3:
a. The verification effort deployed in the current software development process is dominated by dynamic testing typically performed in a sequential fashion on costly (hence, centralised) test infrastructures. This arrangement can hardly sustain the demand for increasingly complex and time-critical systems to be delivered within a dramatically compressed schedule, other than at the cost of higher risks and lesser quality.

b. The software architecture of conventional on-board systems is typically centred around rigid and inflexible cyclic scheduling. The monolithic nature of these architectures makes them ill-equipped to accommodate incremental builds, late changes and development iterations, which are the anticipated characteristics of the future scenario.

The solution space is clearly determined by the objective stated in section 1.2, to place emphasis on the industrial applicability and cost-effectiveness of the proposed solution.

In accordance with these premises, the work presented in this thesis has been directed towards the definition of an engineering approach centred around two major components:

- a verification process geared to static real-time analysis, as a substitute for dynamic real-time testing; and
- an architectural paradigm centred around fixed priority preemptive scheduling, as the evolutionary amendment to the current practice which can attain the combination of flexibility and predictability required of new-generation on-board systems.

The strategic motivation to our work originates from the two-fold objective to:

O.1 attain a more even distribution of the verification effort over a broader portion of the development cycle so as to make it more amenable to the fulfilment of requirement R.1; and

O.2 support a truly iterative and incremental development process which, thanks to the flexibility of the software architecture and the assurance of its predictable real-time behaviour, master the increasing complexity of the software product anticipated by requirements R.2-R.6 within a greatly compressed development schedule.

1.5 Structure and Outline of the Work

The presentation of this thesis is structured as follows: chapter 1 has introduced the problem domain, the objective of the work and the requirements to be met; chapter 2 discusses the distinguishing features of the real-time software embedded on board modern satellite systems; chapter 3 performs a critical review of the current practice for the development of on-board software and identifies the main deficiencies in the face of the emerging requirements; chapter 4 presents the necessary amendments to the current software process, discusses the required enabling technology and presents its current implementation; chapter 5 illustrates the operation of the proposed engineering concept by progressing across the construction of a simple demonstration example; chapter 6 presents the experimental evidence which proves the practicality and performance of
the proposed process model; chapter 7, finally, concludes the presentation by recalling the major motivations, decisions and components of our work, along with an account of the proven accomplishments.

In accordance with the presentation requirements in place at the Technical University of Delft, this document also includes a summary provided in English and in Dutch, a short curriculum vitae of the author and, of course, the bibliography in support of our work.
Chapter 2

Characterisation of On-Board Embedded Systems

2.1 Introduction

Satellite systems orbiting around the Earth, voyaging towards remote points of the Solar System or farther out in the Universe accomplish a broad variety of functions designated to serve user communities characterised by very diverse needs (e.g.: science, business, leisure).

Satellite systems presently serve a growing range of civil applications, which include telecommunications, Earth observation, astronomical, planetary and scientific observation. Some such applications (like, for example, broadcasting or mobile telecommunications) are increasingly commercial and, consequently, face sharper demands for better performance and lower-cost development.

Other applications (like, for example, those aimed at scientific and astronomical observations and / or experiments) remain strictly no-profit and, therefore, tend to stay with more conventional and conservative system architectures for the sake of preserving the safe deployment of the costly scientific package.

In spite of the degree of specialisation required of systems serving so diverse applications, however, the top-level architecture of modern satellite systems is invariably comprised of two main building blocks: the platform and the payload:

- The platform, which may include a number of distinctly separate processing components, collectively performs all functions needed for the control of the spacecraft and the correct deployment of the designated operations. Such functions typically include:

  1. the management of all the communications among on-board components and between the spacecraft and the ground operation centre;
  2. the management of the attitude determination and control of the spacecraft.
• The payload, the actual hardware and software architecture of which ultimately depend on the nature and operation of the designated application, controls the performance of the added-value services or experiments.

Figure 2.1 depicts a principle view of the main building blocks comprised in the architecture of a satellite system.

![Diagram of satellite system](image)

**Figure 2.1: Satellite System Building Blocks.**

Despite the rapidly increasing commercial or scientific importance of the payload, the platform still stands out as the most critical component of the system to the overall economy of the mission. In fact, the achievement of the central objectives of space flight economics, that is: *maximum useful utilisation of all available resources and maximised mission product under all nominal and contingency situations*, critically depends upon the continued provision of essential services by the platform. Only on these grounds can the payload effectively discharge as many of its value-added services as possible. Hence, *the more predictable, efficient and responsive the platform can be made, the more criticality (e.g.: economic value) can be safely accommodated in the payload.*

This thesis discusses development issues and engineering solutions which aim to contribute to increased platform predictability, efficiency and responsiveness, as called for by the requirements discussed in section 1.3.

### 2.2 Overview of Typical Platform Functions

Satellite systems typically perform their operation under direct and continuous supervision by one designated operation centre based on ground. The ground centre is, thus, the single ruling
authority over all of the satellite’s operation. This authority is basically exercised by the issue of telecommands (TC) and the receipt of status and verification information contained in the information or response telemetries (TM) returned to ground by the satellite.

2.2.1 Telecommands

The telecommands sent from ground to the on-board system may carry a broad range of commands to be executed by designated logical or physical devices and/or data relevant to the operation of the satellite. Typical telecommands may enclose:

- simple on/off commanding of selected units and equipment drive electronics (in short, on/off commands);
- requests for immediate execution of predefined command sequences (in short, immediate commands);
- requests for deferred execution of predefined command sequences (in short, event-driven or time-tagged commands);
- data, code and/or new command sequences to be stored in designated memory locations (in short, memory load commands).

[COES, 1994] provides a comprehensive specification of the syntactic format and semantic contents of the set of on-board commands which the European Space Agency use as their standard baseline.

The ability for ground to send telecommands to the in-orbit spacecraft is complemented by the requirement on the on-board system to report on the progress of the processing of the associated commands. [COES, 1994] establishes the central principle of basic telecommand verification: this requires that the on-board design must permit feedback on the execution of any given telecommand at whichever stages are considered meaningful for that telecommand (e.g.: receipt, acceptance, start of execution, end of execution).

The typical up-link bandwidth available for the issue of telecommands ranges between 512 and 2,048 bits per second. Telecommands have a fixed-length header and variable-length contents field (for a maximum theoretical length of 65,542 bytes and a typical size of about 128-256 bytes). Hence, the limited capacity of the typical up-link bandwidth practically means that the on-board system must be able to sustain an inbound flux of 2 to 4 complete telecommand per second. This notwithstanding, the actual operational requirements are generally more demanding than that. The on-board system is, in fact, often required to process the arrival of up to two telecommands per "system cycle", the duration of which is typically correlated to the processing rate of the attitude control activities (50-100 milliseconds). The initial processing of an incoming telecommand typically involves the check for validity, the interpretation of the destination and the storing or forwarding of the associated command information. The arrival of a telecommand from ground must be responded by an acknowledgement telemetry within a fixed response time, which is typically placed within the next system cycle. Similarly, the execution of an accepted
telecommand (or its delivery to destination) must also be accomplished within the next system cycle.

In addition to being the vehicle through which the ground centre exercises control over the operation of the satellite, telecommands also represent the only means for the operation staff to carry out in-orbit maintenance of the system. Simple maintenance operations may, in fact, be performed by commanding the modification of stored configuration parameters. More important corrective, adaptive and/or perfective maintenance operations may require the upload and replacement of portions of the on-board program. Both classes of operations, however designed, require the use of memory load commands. The latter operations, though, are rather infrequent on account of the high inherent complexity and risk. The required modification to the on-board program, in fact, must be performed and verified without discontinuing nor endangering the nominal operation of the system, whereas potentially severe limitations are imposed on it by the communication links between the system and the operation centre. For example, with an up-link bandwidth of 2 kilo-bits per second and a ground visibility of 2 minutes per orbit, it takes at least 3 ground-visible orbital passes (which may be several hours apart) to upload 64 kilo-bytes worth of data.

In spite of the ruling authority residing with the ground-based operation centre and exercised by issue of telecommands, an in-orbit satellite is not entirely passive a system. Two fundamental limitations, in fact, exist which prevent the flow of telecommands and telemetries between the satellite and its ground centre from being continuous and, therefore, require of the satellite system the ability to initiate and perform a varying amount of autonomous operations:

- the orbital trajectory assigned to the satellite may only allow for short periods of ground visibility (e.g.: a few minutes per orbit) and/or severe limitations on the available communication bandwidth (e.g.: very long radio signal round-trip time);
- the mission budget may not afford the cost of a full-coverage operation team and may, thus, need to resort to automated and/or part-time coverage of the mission.

Indeed, the autonomous operation capability of current satellites is still basically confined to the semi-autonomous (i.e. time-tagged or event-driven) initiation and execution of predefined command sequences stored on board prior to launch assembly or uploaded after launch. Such command sequences normally consist of a set of very simple macro-commands written in a mission-specific command language and successively converted (on ground or on board) to a set of primitive on-board functions.

The need for specific command languages as opposed to the direct use of the same programming language used for the implementation of the on-board software essentially originates from the need to allow ground staff, not necessarily educated to the intricacies of software engineering, to be in full control of the nominal operation of the spacecraft. As a consequence of that, the current command languages tend to be rather primitive, mostly unconditional and virtually state-less. This obviously poses severe limitations to the implementation of more advanced autonomous control on board satellite platforms.

The provision of capabilities for increased on-board autonomy is clearly one of the desirable steps towards higher-productivity missions. Initiatives are presently on-going to investigate possible approaches compatible with the current on-board technology. The discussion of the possible
solutions to this particular need, however, falls outside the scope of this thesis. And, in any case, the incorporation of the relevant capabilities in the architecture of the on-board system must be preceded by the implementation of the basic enhancements discussed earlier in section 1.3.

2.2.2 Telemetries

The telemetries sent back to ground from the satellite system are virtually the only means for the ground staff to verify the correct receipt and successful accomplishment of issued telecommands and to maintain up to date knowledge about the current status of the system.

Telemetries are normally sent to ground for confirmation of correct reception of every incoming telecommand and/or successful completion of every required command sequence. Via telemetry, ground has full and continued visibility over the contents of all of the on-board memory and the status of all components.

The typical transmission bandwidth available on the down link for telemetries ranges between 2,048 and 4,096 bits per second and is generally greater than that available for the up-link channel.

2.2.3 Data Handling Control

The management of all of the telecommand and telemetry flow between ground and the in-orbit satellite constitutes one the two fundamental functions of the satellite platform. The services involved in the deployment of these functions include the following:

- telecommand reception, verification and on-board distribution (e.g.: legality checks, priority handling, management and distribution of immediate and/or deferred commands);

- reporting of status information via telemetry (e.g.: confirmation, periodic, event-driven, monitoring, housekeeping and diagnostic telemetries, on-board time for time correlation with ground);

- memory management (e.g.: load from ground, dump to ground, parameter correction, temporary storage of payload data for staggered transmission to ground);

- scheduling of command execution (e.g.: start and stop command sequences, insert, delete and append commands to command sequences, timeshifting of time-tagged command sequences).

This set of communication-related services is collectively termed Data Handling Control subsystem (DHC).

It clearly appears from the earlier presentation in this chapter that the nature of most DHC services is eminently sporadic. These services are, in fact, typically executed in response to communication requests independently arriving from designated sources (whether ground for the issue of telecommands or on-board for the issue of telemetries) or initiated by some on-board resident authority (e.g.: the scheduler of time-tagged commands). The sporadic (as opposed to
aperiodic) nature of these services reflects the fact that the typical operation scenario is expressed in terms of definite mission-specific requirements that dismiss any residual indeterminacy.

In addition to this core of sporadic activities, the DHC subsystem also include a small number of naturally periodic activities such as health monitoring of status parameters and general housekeeping.

It can also be seen that, as a reflection of the centralisation of the down-link and up-link control channels and the fixed communication paths between the on-board and ground components of the space system, most of the nominal DHC activities are deployed in a fairly rigid fashion determined by the precedence relationships functionally established between a fixed battery of data producers and an equally fixed battery of intermediate or final data consumers. The former acquire data from designated data sources (e.g.: the up-link channel; or some on-board data bus). The latter perform transformations on the received data and pass them on to other consumers down the line or to the final data sinks (e.g.: the destination of an on-board command; or the down-link channel).

Analysis of the typical first-order (operational) requirements placed on classical DHC subsystems reveals that inter-arrival times and deadlines can be determined, as second-order requirements, for all of the DHC sporadic activities. Example of first-order requirements would be the maximum flux of TC data to be sustained during nominal operation (e.g.: the ability to complete the initial processing of up to 2 TC every 100 milliseconds – hence, one every 50 milliseconds – and to acknowledge receipt to ground within 150 milliseconds from arrival) or the need to perform oversampled acquisition of selected parameters and periodic dispatch to ground for diagnostic and analysis of on-board anomalies (e.g.: acquisition of up to 64 parameters and issue of up to 2 report TM per second). Example of second-order requirements would be the deadline on the completion time of individual activities participating to a given precedence constrained transaction with an overall bounded response time established as a first-order requirement (e.g.: the need to accomplish the execution of an immediate command or the storing of a deferred command within 110 milliseconds from arrival implies that the execution part of the command has an offset of 50 milliseconds – equivalent to the required completion time of the initial processing – and a deadline of 60 milliseconds).

In view of the need from ground to maintain the most accurate information as to the status and operation of the spacecraft, including time correlation between the request and actual execution of an on-board command, DHC services must in general supply the required response within relatively tight deadlines. Most of the real-time requirements on DHC services can, therefore, rightly be regarded as mission-critical. Failure to meet any of these requirements may, in fact, decrease the mission product (e.g.: when ground cannot efficiently request or command the execution of given on-board actions) or even compromise the mission (e.g.: when ground cannot effectively control the operation of the system).

2.2.4 Attitude Determination and Orbit Control

Complementary to and equally important as the above listed DHC services are the on-board functions aimed at the management of Attitude Determination and Orbit Control (AOC) of the spacecraft.
An orbiting satellite system acquires and maintains its assigned orbital position and trajectory by combined utilisation of a number of sensors (Sun, Star, Earth sensors, gyroscopes) and a set of actuators (torques, wheels, thrusters). In fact, the typical AOC functions are based around periodic acquisition of data from sensors, complex mathematical processing of such data based on specific control laws and determination of corrective actions required of actuators.

In addition to the need for own processing capabilities for the calculation of the required orbital control and trajectory algorithms, the implementation of the basic AOC control functions may need to utilise DHC services for the communication with units located at various points of the spacecraft.

Because of the nature of the applied control laws, AOC activities are mostly periodic with fixed known rates which are desirably set to the highest possible frequency for the sake of achieving higher standards of control performance. With conventional technology, the typical sampling rate of sensor data (i.e. the acquisition) ranges between 50 and 100 milliseconds.

Similarly to DHC activities, AOC activities are naturally structured in a series of producer-consumer clusters organised in a precedence-constrained pipeline structure, as shown in figure 2.2.

![Figure 2.2: Typical Organisation of On-Board Processing Activities.](image)

An AOC control loop typically includes four processing stages, as follows: (1) raw data acquisition from sensors; (2) data transformation for control processing; (3) control processing; (4) commanding of actuators. The need to complete an entire control loop prior to the due start of the next one naturally imposes obvious second-order requirements on the individual activities participating in the process. Such real-time requirements are normally mission-critical in that failure
to meet them may degrade the attitude performance of the spacecraft and ultimately compromise the mission. For example, the deadline for the issue of commands to actuators on completion of the processing of the sensor data is typically set to about 50% of the acquisition rate (i.e. 25-50 milliseconds).

The activity of a satellite system normally progresses across several mission phases, which typically include the following: (a) pre-launch; (b) launch; (c) separation from the launcher; (d) transfer orbit; (e) stationary orbit and nominal operation; (f) de-orbiting or end of mission. The activity of the spacecraft within a given mission phase may then vary across several modes of operation (e.g.: oriented to achieve pointing accuracy during the transfer orbit, such as sun- or earth-acquisition; or oriented to carry out nominal operation in stationary orbit, such as normal mode).

Interestingly, the structure of the software system which governs the operation of the satellite is normally static across mode changes. The code of mode-dependent operation is reached via "switch" decision structures and / or the overriding upload of mode-specific operation parameters. Furthermore, the timing requirements on the execution and completion of mode change operations only call for boundedness and not for instantaneousness. (Which simplifies a whole lot the technical issues involved.)

### 2.3 The Challenge of Integration

In consideration of the limited available processing capabilities and the radically different expertise once required for control and communication engineering, AOC and DHC functions were, historically, allocated to separate subsystems within one and the same platform. Typical platform systems are, in fact, still organised as federated systems, as opposed to classical multi-processor or distributed systems, in which their separate software functions (e.g.: DHC, AOC) are allocated to dedicated and physically disjoint computing "black boxes".

Although the two subsystems would need to communicate and would do so across a very simple serial bus, they would effectively be designed and implemented independently from one another. Consequently, each of them would require a number of common core services so as to be able, in case of need, to take over the survival functions from the other subsystem.

As a matter of fact, since the inception of the 16-bit technology, the federated approach has rarely been chosen as the consequence of feared or incurred excessive CPU load (which seldom exceeds 60% for an individual subsystem). Conversely, this choice far more often reflects the distribution of the required engineering skills.

Despite simplifying aspects of the development and integration process, however, the federated architectural choice is not very cost-effective, for it obviously incurs higher recurring costs and poor mean utilisation of costly on-board resources. Nowadays that global competition drives companies to seek money-saving alliances with former competitors (e.g.: Lockheed - Martin, Boeing - Mc Donald, Douglas, Aerospia - Dassault, etc.) so as to be able to offer lower-cost "turn-key" solutions to the marketplace, the availability of more powerful processor technology such as the ERC32 [ESA, 1992a] is undoubtedly going to question the economic convenience of the federated approach and turn builders to reconsider their choice and possibly transition to
single-computer platforms capable of integrating together the whole set of AOC and DHC functions. We have captured this trend with requirement R.2 in section 1.3.

Such novel systems, which are termed **Integrated Data and Control Systems** (ICDS) are of particular relevance to this thesis as they exhibit the most complex blend of real-time processing requirements to be seen on board current satellite systems. As shown above, in fact, DHC activities are predominantly event- and interrupt-driven *sporadic* with known inter-arrival times, while also including a small number of fixed-rate *periodic* service activities. Conversely, the vast majority of the AOC processing activities are fixed-rate *periodic* with known deadlines.

We have seen earlier that the DHC and AOC subsystems perform time-sensitive mission-critical services. These services, however, have various degrees of criticality. For example, the issue of some "routine" TM by a subsystem (e.g.: those carrying cyclic health status information) may sometimes exceed the required production deadline without this seriously endangering the safety of the spacecraft, whereas most of the incoming TC normally require immediate attention by the subsystem and have to be serviced within guaranteed response time. Actually, DHC and AOC systems also discharge a few *non-critical* activities which can be performed in the background under comparatively relaxed timing requirements. A typical example of non-critical background service is that which periodically scrubs the entire RAM bank of the platform computer as a means for protection against the occurrence of radiation-induced double bit flips on the same memory cell. This service is useful because normal EDAC (Error Detection And Correction) devices can only protect from single bit flips, and hence a periodic EDAC-protected read / write cycle of all the memory cells completed within a finite time lowers the probability that a second bit flip hit the same memory cell before the first one had been corrected.

In summary, we can say that conventional ICDS systems would definitely belong in the category of mission-critical real-time systems. ICDS systems would in fact typically include a mix of *hard* (i.e. mission-critical) and *soft* real-time components characterised by *periodic* and *sporadic* activation requirements, and a large component of *non-independent* and *precedence-constrained* activities.

![Figure 2.3: Principle Architecture of Ideal ICDS System.](image)

The "entry" size of ICDS systems (i.e. only inclusive of conventional DHC and AOC functions) would typically amount to 25,000 source lines of code, thereby placing at the high end of present generation systems, and incur a processing load in the range 50% to 80% on space-qualified 16-bit technology. The principle block diagram of an ideal ICDS system, with a broad indication of the included functional components, is depicted in figure 2.3.

The demand to increasingly move criticality from the ground centre onto the spacecraft platform is one of the means to reduce the operation cost of a space system. This provision will require new-generation ICDS systems to incorporate a host of novel value-added functions, such as, for example:

- **resource management** services, for the planning of operations and mode transitions aimed at the minimisation of resource consumption (e.g.: fuel, power, memory) and the maximisation of mission product;

- **mission management** services, for the activation, execution and control of mission phases and mode-dependent operations, in autonomy from ground control;

- **on-board diagnostic** services, for the accomplishment of early localisation and diagnosis of on-board anomalies prior to the involvement of ground staff.

These additional capabilities will be progressively added to the spacecraft in successive increments along with the consolidation of their functional and operation requirements. Hence, the architecture of the on-board software must be flexible yet predictable enough to accommodate additional functional components without incurring inordinate increase of implementation complexity nor operation hazard.

2.4 Summary of Important Characteristics

This chapter has provided a short overview of the typical software functions performed by platform systems. The discussion has highlighted the most distinguishing properties, needs and requirements of the typical on-board control services and shown that, especially in view of the emerging requirements presented in chapter 1, the construction of the software component of such services presents the designer with important demands and challenges.

The discussion so far has allowed us to determine that modern platform systems are:

- **remotely operated**, interacted with only by telecommand / telemetry and maintained exclusively by selective upload of software upgrades;

- **long-lived**, built with material exposed to rapid technological obsolescence and, owing to their relatively long development cycle, hardly suited for practical reuse in subsequent (often several years apart) flights;

- traditionally **one-off**, as a further hinder to the achievement of any conventional "reliability by use" evolution; now gradually evolving, in some specific commercially exploitable
sector such as mobile telecommunication, towards more conventional product / family concept (like, for example, in the case of the commercial "Globalstar" and "Iridium" programs [Miller, 1998], which aim at the staggered launch and simultaneous operation of about 50 identical small satellites supporting world-scale mobile telephony);

- dependent on the use of specialised hardware material and special production process inevitably exposed to high cost, low investment and slow technological advance;

- increasingly software-intensive, as shown in figure 1.1.

Within this context, this chapter has shown that modern on-board platform control systems include the following software components:

1. predominantly event- and interrupt-driven sporadic DHC activities, characterised by known minimum and maximum inter-arrival times;

2. known fixed-rate periodic AOC and DHC processing activities;

3. time sensitive mission-critical services with differing degrees of criticality, known completion time requirements (deadlines) operated in a producer-consumer pipeline-type functional structure as shown, for example, in figure 2.2;

4. distinctly non-critical self-standing background activities;

5. a number of centralised services (e.g.: TC reception, TM issue, command management) used by the most part of DHC and AOC activities (which are, therefore, obviously non-independent and often related by definite precedence activation constraints).

In force of these characteristics we can say that the software systems embedded on board satellites do belong in the class of real-time systems in general and to that of mission-critical real-time systems in particular.

The programming model proposed in chapter 4 shall obviously prove adequate to support systems with these characteristics. Moreover, it shall also provide for the extent of flexibility and predictability required to effectively accommodate additional functions aimed to increase the value added of the platform.
Chapter 3

Weaknesses of and Amendments to the Current Software Process

3.1 Introduction

Chapter 1 discussed the challenge presented to the current software process and associated technology by the requirements placed on new-generation on-board software systems. The discussion highlighted the aspects of the process and product which will be most affected by the emerging requirements and outlined the solution space within which we want to determine the necessary enhancements. This chapter takes the discussion further and introduce the key elements of the proposed solution which will be subsequently elaborated in chapter 4, demonstrated in chapter 5 and finally evaluated in chapter 6.

The discussion thread in this chapter progresses from the presentation of the reference context, via the interpretation of the needs, on to the establishment of the envisaged solution and the positioning of our interpretation with respect to other practices in the same application domain. In particular,

a. section 3.2 (the presentation) outlines the current software process and the main features of the reference technology in use at European space industry;

b. section 3.3 (the interpretation) analyses the deficiencies of the current software process and associated technology in the face of the requirements discussed in chapter 1;

c. section 3.4 (the solution) establishes the solution space within which the required amendments to the software process and associated support technology are to be defined; the section also includes an initial presentation of the main guidelines which have inspired the development of the proposal we will discuss in chapter 4;

d. section 3.5 (the positioning) finally, relates the process and technology that we address in this work to other mainstream practices in the same application domain.
3.2 The Present Status

3.2.1 Software Process

The development of on-board embedded software is but one part of the overall process leading to the development of a satellite system. The life cycle of a satellite development project is typically comprised of the following four sequential phases:

**Phase A: Concept Definition.** This phase encompasses the definition of the technical and strategic objectives of the project.

**Phase B: System Definition.** This phase includes the analysis of alternate approaches to the accomplishment of the project objectives and the definition of the desired system concept.

**Phase C: System Design.** This phase comprises the establishment of the system development schedule and the associated management and procurement plans. The phase includes the system break-down and the apportionment and specification of the resulting subsystems. (The overall system concept is normally comprised of the satellite itself, typically referred to as the "space segment", as well as of the ground-based control station, typically referred to as the "ground segment").

**Phase D: System Development & Operation.** This phases embraces the development of the subsystems identified in phase C, their integration, qualification testing and final acceptance by the contractual customer.

The typical duration of the phases C and D (i.e. the satellite development proper) presently ranges between 3 and 4 years. Requirement R.1 in chapter 1, though, has highlighted the demand to compress this duration down to 18-24 months.

The life cycle of the software embedded in the space segment of the system spans across the late part of phase C and the early part of phase D of this schedule. The latest delivery time of the baseline software product for PROM burn-in and installation on the flight hardware is typically placed 4 to 6 months ahead of the final acceptance at satellite level.

The software development model currently in place at European space industry is governed by various instantiations of the PSS-05 standard regulation [ESA, 1991]. This regulatory standard was emanated by the European Space Agency in 1991 as the recommended practice for use by European space industry and since achieved generalised acceptance as reference model [Jones et al., 1997].

The development process informed by this standard is normally referred to as the "V model", in recognition of its V-shaped graphical representation. The PSS-05 model is in fact comprised of one descending branch, which embraces the definition and implementation phase, and an ascending branch, which departs from the tail of the preceding branch to include the integration and verification phase. Figure 3.1 depicts the logical flow of the PSS-05 development phases and indicates the relative proportion of effort typically absorbed by the main segments of the process.
The *descending* branch includes the following activities:

**User Requirements (UR) Definition.** The contractual customer issues the system specification in the form of a User Requirements Document and a corresponding Software Verification & Validation Plan for Acceptance Testing (SVVP / AT).

**Software Requirements (SR) Definition.** The specification given in the User Requirements Document is transformed into a *logical model* of the system and describes this model in the Software Requirements Document. This phase includes the definition of the V&V plan to be executed at software system integration level (SVVP / ST).

**Architectural Design (AD) Definition.** All the implementation decisions required to turn the logical model of the system into a concrete *physical model* are documented in the Architectural Design Document. This phase includes the definition of the V&V plan at component integration level (SVVP / IT).

**Detailed Design (DD) and Unit Testing (UT).** All individual components of the software system are designed to completion, coded and functionally verified at unit level in accordance with the V&V plan defined for this purpose (SVVP / UT).

![V-model Development Cycle Diagram](image)

Figure 3.1: The V-Model Development Cycle.
The *ascending* branch, globally referred to as the Transfer phase, includes the following activities:

**Component Integration Testing (IT).** Focus is placed on the verification of the functional behaviour at the interface between individual components of the system; it is at this stage that the software under test is first installed on a target-representative environment.

**Software System Integration Testing (ST).** The individual components are integrated into a completely operational system and the correct operation is verified in accordance with the plan previously defined in the SR phase.

**Acceptance Testing (AT).** The software system is installed in the designated operational environment and the acceptance tests defined in the SVVP / AT are executed, often at the presence of the contractual customer. The AT phase marks the completion of the software development process and is normally followed by the *Operation and Maintenance* phase where the accepted software product is used in the operational environment by the end user.

### 3.2.2 Support Technology

No process exists in practice without an associated enabling technology. The definition and introduction of the PSS-05 standard were, thus, accompanied by the development of those technology elements which were, at the time, perceived as missing from the commercial market. As a result of that, and as a reflection of some genuine needs of the space domain, the enabling technology associated with the PSS-05 standard did end up including a combination of space-specific and general commercial technology. The relative freedom from the variability of the commercial technology which ensued from that choice did in some sense grant space industry the technological stability necessary to develop a considerable experience and a vast legacy with the chosen reference technology.

Irrespective of its origin, though, any enabling technology does inevitably exhibit features and limitations which determine features and limitations of the software process it aims to support. Hence, in order to better capture the deficiencies of the current process in the respect of the emerging requirements, it is necessary to take a closer look at the main technology elements presently in use.

In particular, we examine those which support the *descending* branch of the PSS-05 model. This will allow us to understand what sort of system is typically presented to the testing phase, which, as we have seen, presently takes up the most part of the typical development effort.

As a result of decisions made in the early 80s, the software technology most widely used by European industry in the specification, design and coding of on-board software is centred around the combined utilisation of the HOOD (Hierarchical Object Oriented Design) design method [HTG, 1993] for the design and the Ada programming language [ISO, 1987] for the implementation. In fact, virtually all of the software systems for use on board European satellites (whether for platform or payload utilisation) have been (or are being) designed using the HOOD method. At
the same time, [Ratcliffe, 1995] reports that, up until the end of 1994, over 90% of the on-board software flying or being developed for use on European satellites was being written in Ada; a trend which has continued until the present date without any noticeable variation.

As the Ada technology is sufficiently known to the intended readership that it can go without any specific presentation, in the following we only briefly outline the main features of the HOOD method.

Overview of the HOOD Method

HOOD was originally defined in 1987 as a result of an industrial effort funded by the European Space Agency and ever since matured into a standard method through continued usage in most of the Agency-funded software projects.

The definition of the method was deeply connected with the formation of the PSS-05 development model and specifically intended to cover the AD-DD segment of the V model shown in figure 3.1. The HOOD method is presently configured as a standard and maintained by a User Group.

Commercial implementations of the latest standard version of the method industrially applied (i.e. version 3.1) are available and presently in use by most space companies in Europe. Numerous instances of utilisation of the method are also reported from other domains of application (e.g.: transport, energy, nuclear).

In its original definition and up until version 3.1 of the standard, HOOD is, in spite of its denomination, to be regarded as an object-based architectural design method which results from a combination of abstract machines and object-oriented design concepts. As it provides for a graphical representation of the system as well as for a textual language, HOOD can also be regarded as a structured design method.

Similarly to any other object-based development method, a HOOD system is viewed as a collection of co-operating objects which can be hierarchically decomposed. The allowed decomposition process, however, does not support class hierarchies and full (i.e. multiple) inheritance and, therefore, cannot be truly called object-oriented. These limitations may probably be attributed to the fact that the original definition of the method directly targeted Ada 83 [ISO, 1987], which was object-based and not object-oriented, as the preferred implementation language.

The recent advent of truly object-oriented variants of the two sole implementation languages considered by [ESA, 1996] of immediate interest to Agency-funded developments, that is: Ada 95 [ISO, 1995] and C++ [ISO, 1997], has prompted a new revision of the HOOD standard (denominated HOOD 4) [HOOD Users’ Group and Rosen, 1997], which aims at being truly object-oriented and providing for direct code extraction mechanisms targeted to both languages.

No actual usage of HOOD 4 is presently reported, though, and none is expected until mature tool support will become commercially available. In the remainder of this document, therefore, we shall exclusively consider the version of the method which is actually in use at industry, that is version 3.1. Hence, all occurrences of the term HOOD in the following shall be understood to denote HOOD 3.1.

The HOOD method is a hierarchical design method. With HOOD, the software system is initially designed as a single object denominated the root object. The root object is then decomposed
into child objects. Child objects are subsequently treated as either parent objects and, hence, further decomposed into new child objects, or terminal objects, which cannot be decomposed any further.

The method supports direct inheritance between parent and child objects. Hence, child objects inherit the properties and constraints of their parent object. The combination of hierarchical decomposition and direct inheritance enables the HOOD method to support the controlled partitioning of portions of the system design to any number of hierarchically organised development partners.

HOOD objects are defined by the service they provide for use by other objects. The provided service is specified in the interface of the object. The internal details of the operation of the object are described in operation control procedures. The behaviour of the object with respect to service requests received from other objects is described in the object control structure.

Two types of relationships may exist between HOOD objects: (a) the parent-child relationship, which is denoted by the "include" (or "implemented by") relationship; and (b) the "use" relationship, which denotes that an object requires the services of another object.

The operations exported by an object are associated with execution requests which are issued from a client object for execution by the server object in charge. Operations inside an object are activated by the triggering of control flows within the object. Several threads of control may operate simultaneously within an object in accordance with the type and number of the provided operations.

The execution request for operations provided by HOOD objects can be as follows:

unconstrained : when control flows from the client into the server for execution of the operation and returns to the client upon completion of the operation;

state-constrained : when the execution of the requested operation is conditioned to the internal state of the server as determined by previous execution;

asynchronous (ASER) : when control flow in the client is not interrupted while the requested operation is triggered within the server;

highly synchronous (HSER) : when control flow in the client is interrupted until completion of the requested operation within the server;

loosely synchronous (LSER) : when control flow in the client is interrupted until the server acknowledges receipt of the execution request;

timed-out highly synchronous (TOER) : as for HSER but control needs to be returned within a given time out.

Both the unconstrained and the state-constrained operations do not cater for parallel execution of the client and the server flow of control, while all other operations do. As a reflection of that, the HOOD method differentiates between two types of objects, depending on how control flow takes place in them:
• **passive** objects, which have no internal concurrency (i.e. provide for purely sequential control flow for they possess no control flow of their own) and, hence, can only provide for unconstrained and state-constrained operations;

• **active** objects, which have a thread of control of their own and can be, therefore, used to express concurrency; hence, active objects must provide at least one constrained (but not state-constrained) operation.

Besides the above, the method does not introduce any other prescriptions to aid the designer in determining or constraining the desired control structure of the system.

### 3.3 Cheaper, Faster, Better

The introduction of the PSS-05 standard was primarily intended to facilitate the production of verifiably higher-quality software products within a controllably reduced cost envelope. A sufficient body of evidence exists nowadays to prove the achievement of some important progress in that direction.

As suggested by the discussion in chapter 1, however, much remains to be done for space software to stay a cost-efficient ingredient of new-generation software-intensive satellite systems in the face of the emerging requirements. In fact, the rise of the "cheaper, faster, better" mission paradigm which we have discussed in chapter 1 will cause the future scenario to be dominated by the demand to:

1. compress the satellite development schedule from the present 3-4 years to 18-24 months [ESA, 1995b, Van der Ha, 1995];

2. deliver better mission product via increased autonomy and responsiveness of operation [Teston et al., 1997];

3. support the trend to increasingly more software-intensive systems, the size of which is predicted to rapidly rise from the current 15-25,000 source lines to the 50-60,000 of new missions under the same language baseline [ESA, 1996, Vega, 1993]

A principle analysis of the fitness of the present industrial practice in the face of the emerging requirements yields the following considerations:

• as reported in section 1.3, data available from a collection of recent Agency-funded satellite development activities consistently indicate that the software productivity rate measured across the entire sequence of the PSS-05 software development phases (i.e. the SR–ST segment in figure 3.1) ranges between 0.75 - 1.0 delivered source lines per hour for on-board systems typically comprised of 15-25,000 source lines: it is apparent that the productivity figure needs to at least quadruple (on account of twice as big software to be developed in half the time) for the cost of new systems to remain economically viable;
• common wisdom has it that the effort, time and costs associated with the descending branch of the PSS-05 V model (i.e. the SR-DD segment) are normally well mastered so long as the user requirements are stable enough across the entire development process; in fact, this is rarely the case and it will be even less likely so in the future, for the required reduction of the development schedule will not tolerate the long latency commonly allowed for requirements analysis and consolidation: hence, it is more exact to say that the descending branch of the PSS-05 model is fragile against incomplete or unstable user specifications;

• available evidence shows that the ascending branch of the PSS-05 V development model – which encompasses the testing phases from component integration (IT) to software integration (ST) –, typically absorbs in excess of 50% of the total software development time (cf. figure 3.1 for the typical relative cost proportion of the main development phases); its tail position in the process and its effort-intensive conduction expose the IT-ST segment of the V model to the largest risk of delay and cost overrun; it is clear, therefore, that the productivity of the testing phase is very seriously challenged by the increasing size and complexity of new-generation systems and the urge to greatly compress the development schedule.

These observations clearly denote the demand for important improvements to the overall efficiency of the current software development process. In particular, we assert that the present development model suffers from two major deficiencies: (i) the ill-balancedness of the effort distribution; and (ii) the overall inadequacy of the current practices in the face of the emerging requirements. We base this assertion on the following observations:

• Ill-balancedness: the testing activities in the ascending branch are presently dominated by dynamic tests performed on the physical target. These activities possess a centralised and sequential connotation which obstructs the reduction of their duration without detracting from the ultimate quality of the product.

• Inadequacy: The effectiveness and coverage of the testing phase critically depend upon the quality of the process performed in the descending branch. We concur with the argument developed by [Matra Marconi Space UK, 1995], however, that the current practice in the descending branch is insufficiently equipped to master the increasing real-time component of new-generation systems. The urge to compress the development schedule would thus increase the risk that the ascending branch be presented with a loose definition of the system, especially in the regard of the expected real-time behaviour. This phenomenon will be further amplified by the drastic reduction of the extent of real-time verification attainable by dynamic testing in the future scenario.

We argue that these weaknesses largely emanate from deficiencies of the descending branch and will substantiate this claim later in this chapter.

Let us now first look into the reasons behind the ill-balanced effort distribution, for this is appears to be the most indicative symptom of the problem we want to cure.

In actual fact, there may be several reasons why the testing phase takes up the largest proportion of development effort. This thesis recognises the following two primary causes:
1. intrinsic limitations with the available test support technology may greatly impair the productivity of the testing phase;

2. improper load is frequently transferred to the testing phase from deficiencies of activities situated in the descending branch of the V model.

Whereas the former aspect is broad enough to form the subject matter for a whole new research line and, as such, falls outside the scope of this thesis, the latter aspect is of specific interest to our discussion.

The basis to the special interest in that aspect lies in the assertion that the present "25-15-60" scheme of effort distribution reflects the fundamentally duplicitous nature of the PSS-05 standard. The PSS-05 standard, in fact, assumes an essentially sequential development model, whereby individual phases are entered and exited only once (while they may be internally iterated any number of times). This model has a strong managerial (in fact, contractual) connotation and supports a rather purged view of the development progress. This view obliges to the management ambition that the ascending branch be entered (and the associated contractual milestone achieved) irrespective of there being outstanding aspects left unresolved from the descending branch.

The obvious down-side of this model is that it surrenders too easily to the temptation of pushing forward to the next phase as many of the outstanding "open" aspects of development as the letter of the standard allows. Real developers know that this phenomenon, which we wish to nickname the "snow-plough" syndrome, is not rare an event.

In essence, this thesis asserts that, as a result of the "snow-plough" syndrome, deficiencies in the operation and organisation of the activities in the descending branch of the V model would tend to inflate the effort and complexity of the activities in the ascending branch with an improper load of deferred issues. In the following, we identify and analyse such deficiencies.

### 3.3.1 Deficiencies of the Software Requirements Definition Phase

The Software Requirements Definition phase aims at capturing the functional and non-functional requirements which derive from the user specification and determine the logical characteristics of the chosen solution (assembled in the so-called logical model). The former category of requirements define what the software has to do. The latter identify the constraints within which the software is to accomplish the required function and, consequently, include the real-time requirements established on its operation.

The taxonomy of non-functional requirements defined by the PSS-05 standard includes the following:

**Quantitative Requirements**:

- *Resource Requirements*, which specify upper limits on physical resources.
- *Performance Requirements*, which specify quantitative values for measurable static (e.g.: capacity) and dynamic (e.g.: utilisation, throughput, duration) performance values.
Example of Typical Performance Requirements:

- The On-Board Software (OBS) shall have a maximum CPU utilisation not greater than 75% in the worst case.
- The OBS shall reset the Watch-Dog no later than every 100 millisecond.
- The OBS shall be able to sustain a maximum telecommand flux of 500 bit/s.
- The OBS shall be able to sustain a maximum telemetry flux of 2,500 bit/s.
- The OBS shall initiate the generation of a Report telemetry packet within 150 millisecond after receipt of a telecommand.

- Interface Requirements, which specify the physical and logical elements which the software system is to interact with.
- Operation Requirements, which specify how the software system is intended to be run and operated (human-computer interface, logistics, organisational aspects).

Qualitative Requirements:

- Verification Requirements
- Acceptance Requirements
- Documentation Requirements
- Security Requirements
- Portability Requirements
- Quality Requirements
- Reliability Requirements
- Maintainability Requirements
- Safety Requirements.

Real-Time Requirements

According to the above definition, the real-time requirements are to be regarded as non-functional quantitative requirements. Alike all other kinds of requirements, the PSS-05 standard expects that they also be fully characterised on exit from the Software Requirements Definition phase.

It is commonly experienced in the development of an on-board embedded real-time systems, though, that a great deal of real-time requirements tend to arise as second-order requirements which the designer derives from translation of direct user requirements in accordance with the adopted physical model of the system.
This consideration reveals an important deficiency of the current PSS-05 development model when applied to on-board embedded real-time systems: whereas the software requirements definition phase is supposed to be solely concerned with the establishment of one logical model of the system which meets all of the known requirements on the system, it appears that the comprehensive definition of all of the quantitative requirements (hence, both direct and derived) does indeed require the designer to anticipate decisions (features, properties and constraints) which emanate from the physical model of the system.

Hence, it is clear that if the definition of the physical model is deferred until completion of the architectural design phase (which is the normal course of action) second-order real-time requirements may continue to arise well after the nominal completion of the requirements definition phase. This exposes the development to the risk that important requirements may escape proper formalisation, analysis and tracing as well as the definition of an appropriate verification strategy at software integration test and beyond.

Chapter 2 has briefly characterised the distinguishing features of software systems embedded on board satellite systems. This has shown that the real-time embedded nature of these systems does indeed require that the operational correctness of the system be systematically verified in the time domain as well as in the value domain.

This concern has important consequences on the requirements put on the design and implementation of the system and, consequently, ought to be better served by the development method. In particular, as the discussion so far has highlighted, it calls for greater and more systematic attention to the manner (i.e. formalism and means) in which the entire set of first- and second-order real-time requirements are captured, expressed and verified. Any amendment to the current software process motivated by the urge to meet the emerging requirements should address this deficiency and avoid that design and implementation iterations occur uncontrolledly in the IT-ST segment of the V model.

3.3.2 Deficiencies of the Design Definition Phase and Technology

On exit from the SR phase, PSS-05 designers enter the AD phase where they would use the HOOD method to express their design. The method would then be employed to cover the entire AD phase and a variable portion of the DD phase, the extent of which would normally depend upon the quality of the support provided by the HOOD tool of choice.

The HOOD method and the associated technology are certainly not exonerated from shortcomings and limitations. In particular, this thesis contends that the HOOD method has one major deficiency in the face of the emerging requirements discussed in chapter 1: while new-generation systems become increasingly concurrent and time-critical, in fact, the method does only allow for a loose expression of concurrency and provide little conceptual support to describe the control structure of the system.

As outlined in section 3.2.2, HOOD does indeed contemplate basic rules to allow the designer to determine how threads of control are allocated to individual objects. The method, however, supplies no binding prescriptions to determine the rules which are to govern the concurrent execution of threaded objects at run time. Consequently, the definition and verification of the real-
time behaviour of a HOOD system would only be addressed on generation and execution of the program.

The lack of any definite binding prescription to the execution platform of choice is one of the factors which affect the quality and effectiveness of the support provided by the HOOD technology for the coding phase. As a result of that lack, in fact, the construction of an executable program from a HOOD design does always require some form of hand-coded and user-specific binding to the run-time component of the execution platform of choice. The criticality of that effort obviously increases with the complexity of the application. The effect of this limitation has thus been to ultimately prevent the method from delivering any worthwhile contribution to the coding phase.

One cannot prove what one has not defined. Consequently, the combined effect of this limitation of the HOOD-based design definition phase together with the deficiency of the process discussed in section 3.3.1 would basically have the consideration of the real-time aspects of the system deferred well into the ascending branch of the V model.

The net effect of that deferral, however, is merely to offload concerns from the earlier phases of development only to shift them towards the test phase. Thus, the test phase becomes the “catch-all” bucket for all of the aspects related to the execution behaviour of the system.

That may perhaps be a convenient paradigm for non-real-time systems. Certainly, however, it is not the best avenue for the development of new-generation time-critical on-board systems. In fact, it does ultimately add an unnecessary extra load to the already high level of effort and complexity which is presently required of the test phase.

The above is clearly one of the main reasons for the ill-balanced distribution of effort across the various segments of the V model and another indirect effect of the "snow-plough" phenomenon discussed earlier. We contend that this deficiency of the design definition phase makes the present interpretation of the V model extremely fragile in the face of the increasing real-time complexity of new generation on-board systems and should be therefore accordingly amended.

3.4 Solution Space

3.4.1 Evolutionary versus Revolutionary Amendments

In the continuous quest for methods and tools which can improve the productivity of the process for the development of on-board embedded real-time software, both industry and the European Space Agency have recently started investigating a number of techniques which promise important breakthroughs in that direction. A non-exhaustive list of the "hottest" candidate techniques includes the following:

- formal description techniques, behavioural modelling and executable specifications, as the means to a faster and more reliable specification phase;

- automated code generation, as a means to a faster production phase;

- extensive software reuse, as a means to increase the productivity of the process and the reliability of the product.
It is known for a fact that initiatives aimed to prepare for the introduction of selected candidate techniques in the industrial development process are already in progress within several companies in Europe. What presently remains unclear, though, is which modifications shall eventually be required of the current standard development process and how best (i.e. swiftly, smoothly and economically) can the associated technology be brought up to proficient utilisation.

The introduction of any novel techniques within the current software development process can be accomplished in an either evolutionary or revolutionary fashion. The latter approach would prompt the progressive dismissal of the present software technology and process and replace them altogether with "foreign" ones which proved capable of successful exploitation of the favoured novel technology in other, possibly similar, application domains (e.g.: fly-by-wire aircraft). The former would retain the good value of such methods and tools as HOOD and Ada, enhance them as required and have them possibly co-operate, rather than conflict, with the selected novel techniques.

Both approaches are equally worth of consideration and, in fact, at present, there does not appear to be any distinct consensus on either option. There do exist, however, a good number of technical and non-technical reasons for this thesis to advocate the adoption of the evolutionary approach, at least in the short-to-medium term period.

Technical reasons in favour of the evolutionary approach find support, for example, in the evidence supplied by a study performed by the Software Engineering Laboratory over the software practice at the National Aeronautics and Space Administration (NASA). The results from the study were originally published in November 1993 [SEL, 1993] and recently reconfirmed by [Waligora et al., 1997]. The cited study reports that the combined utilisation of Ada and object-oriented design techniques in the production of a variety of software systems for space application (both ground and on-board) assessed across the equivalent of the SR–ST segment in figure 3.1, was proven to have:

- greatly improved the extent of software reuse (+300%) and the software productivity (from an average of 1.1 source line per hour to about 2.5)

- significantly reduced the overall system development costs (-40%) the turn-around time (-25%) and the residual error rate (-62%) in comparison with earlier developments centred around the use of Fortran and functional specification techniques.

Even though similar data are not homogeneously available for the European Space Agency, we trust we can safely assume that European industry should have observed comparatively similar results from the combined utilisation of HOOD and Ada over the last decade.

Equally important non-technical reasons in favour of the evolutionary approach arise from the known existence of important cultural and economic barriers to sudden revolutionary changes. These barriers include, for example, the momentum of mature exploitation of technology in large industrial organisations; the untenantable cost of repeatedly training personnel to proficiency with a given technology; and, last but not least, the considerable legacy of HOOD and Ada systems and knowledge built up to the present date [Ratcliffe, 1995].

All this considered, the choice of the evolutionary approach is the direction favoured by this thesis. This choice, which places in full keeping with the programmatic directions set out in

[ESA, 1996], retains and builds upon HOOD and Ada as the "given" reference technology in the construction of the proposed technical solution.

3.4.2 Scope of the Proposed Amendments

The effort-distribution figures shown in figure 3.1 suggest that the required modifications to the current software process should in preference attack the cost of the ascending branch of the V model, for that is presently more labour-intensive than the descending branch.

The discussion in section 3.3, though, has argued that the present ill-balancedness is, to a great extent, the symptom of distinct deficiencies of the descending branch. Accordingly, our work aims to cure the disease rather than solely the symptom. Hence, the research direction of this thesis set out in section 1.4 and the rationale discussed earlier in this chapter call for the adoption of evolutionary methods and techniques capable of increasing the efficiency of the descending branch by advancing useful elements of verification to earlier phases of development. This will be accomplished by the introduction of novel techniques for the engineering and verification of the real-time requirements placed on new-generation on-board software, in the context of a dramatically compressed and more productive development process.

Our aim is to spread the load of concerns typically charged to the ascending branch over a broader portion of the development process. More specifically, we want to replace an important amount of dynamic real-time testing occurring late in the development process by a functionally equivalent amount of static real-time analysis occurring earlier in the process.

Dynamic testing is performed late in the process on representative, hence costly, platforms and consequently executed in an inherently sequential and centralised fashion. Conversely, static analysis can be performed earlier in the process and needs no costly centralised support. Intelligent use of static analysis will therefore lessen the labour-intensiveness of the testing phase and increase the likelihood that work already performed may be preserved from the disruption caused by the introduction of late structural changes.

In order to achieve this objective, however, we need to make sure that the extent of real-time verification performed at the design level be not later undermined by uncoordinated implementation decisions nor inadequate implementation means. At the same time, we also need the ability to iterate over the design definition and verification steps without this causing intolerable penalties nor delays to the implementation.

The choice, definition, implementation and assessment of the techniques which we will propose in chapter 4 and 5, thus place within the solution space inscribed within the following axes:

- adoption of the evolutionary approach for the exploitation of software development techniques aimed at the required cost reduction (i.e. enhanced utilisation of such methods and tools as HOOD and Ada, in conjunction with the housing of the selected novel techniques);

- adoption of a development approach which aims at mastering the belated consolidation of (functional, operational and real-time) requirements which may be anticipated to result from the required reduction of the development schedule.
In order to stress the strategic importance of the latter axis, it should be noted that [SEL, 1993] models the "growth factor" incurred by the software development costs as a result of late consolidation of specification – the notorious "TBD" effect –, with a coefficient of 1.4: that is, an unstable specification may cause the development to incur a cost increase of as much as 40%. That consideration strongly reinforces the earlier argument about the potentially negative effects of the "snow-plough" syndrome introduced in section 3.3. Furthermore, it shows how essential it is that the development process be equipped with the ability to explicitly contemplate and support the occurrence of various instances of design and implementation feedback incurred in the course of the development.

The following section shall introduce the main guidelines which have inspired the definition of the proposal presented in chapter 4 and 5.

3.4.3 Proposed Amendments to the Current Software Process

The PSS-05 standard, in its current version, introduces no differentiation between on-board and ground applications. With time, however, it has become increasingly apparent that this was a deficiency rather than a virtue, for the development of on-board embedded real-time software does indeed differ in some important respects (such as, e.g.: the execution environment, the development logic, the predominantly one-off nature) from that of conventional software.

The acknowledgment of such a deficiency and the proposition of suitable amendments to the PSS-05 process model in recognition of the needs and properties of new-generation on-board embedded real-time software systems have recently been addressed by an Agency-funded research project entitled REALISME [ESA, 1995a, E2S, 1997]. The author of this thesis actively participated in the conception, conduction and conclusion of the REALISME project in all areas of the project which were of interest for this thesis. The findings presented in this chapter are broadly in line with the observations made in [Matra Marconi Space UK, 1995] by REALISME, but go much further as to the analysis of the problem domain and the definition of a practical engineering approach to the construction of new-generation systems.

Similarly to the approach advocated by Kopetz et al. in [Kopetz et al., 1993], we maintain that the issues of timeliness and synchronisation involved in the construction of a real-time system are to be addressed at the software design level, rather than treated as an implementation-specific exercise or else imposed as a system-level view of the software architecture.

In PSS-05 terms, the software design level corresponds to the establishment of the physical model. Hence, our goal is to augment the power of expression and verification availed to this level, so that the real-time structure of the system can be established, consolidated and verified within the physical model, while leaving the implementation and verification of all functional aspects to the subsequent phases of development.

In contrast with [Kopetz et al., 1993], however, we want to acknowledge and facilitate the occurrence of feedback-based development iterations (as the basis to the incremental consolidation of the system) across a greater portion of the life cycle than the MARS approach allows.

In order to accomplish this, we need to build on the following three conceptual ingredients:

L.1 A unified design framework, which enables the reconciliation of the complementary con-
cerns of the logical and physical model of the system into a coherent and consistent overall model; the framework is to achieve this by facilitating feedback and iteration between the definition and implementation of the logical and physical model of the system; the framework is the development space in which the designer consolidates and commits the system concept, verifies the feasibility of the physical model in the time domain, and builds on the relevant evidence in the construction of the system integration and verification strategy. This provision will extend the span of the process devoted to the definition of the physical model past the conventional confines of the design phase.

L.2 A well-defined Computational Model, which encompasses the definition of: (a) the components which can be used in the construction of the real-time architecture of the system, the means availed for the communication and synchronisation between them, and the concurrency paradigm assumed for their run-time execution; (b) the execution properties and real-time attributes which characterise those components as well as the constraints placed on their use and interaction; and (c) an underpinning analytical model which allows the static analysis of the real-time behaviour of systems constructed in terms of those components. The collection of these definitions constitute the abstract view of the architectural support for predictability availed to the software system design phase. The details of such definitions may vary with the particular Computational Model of choice and may require specialised hardware support, especially if important dependability requirements on the system are outstanding.

L.3 The realisation means, which enable the enforcement of the "contract" established between the Computational Model and the programming interface to be used for its implementation. The ability to enforce this contract is a critical element of our strategy, particularly as the implementation may rely on constructs and services of a general-purpose programming interface (whether language or operating system based) which can break the integrity of the Computational Model of choice. The required extent of enforcement can be executed in several ways and at several levels. For example, at the level of the compilation and linking process (by denying access to illegal primitives and constructs); at the level of the design (by providing for a direct and automated translation to a legal implementation); or else by a combination of static tools, like in the case of the SPARK Examiner [Carré et al., 1986, Carré and Garnsworthy, 1990].

The concept of Computational Model stands central to our development strategy. The formalisation of this notion configures as an original contribution of our work, which takes further the initial definition given in [E2S, 1997].

We now want to discuss in more detail the exact scope and function which the concept of Computational Model encompasses in our construction.

The name we have chosen echoes that of Programming Model, which is the notion commonly used to denote a certain subset of the constructs and means supported by the programming interface associated with any given implementation language (or its binding to any given operating system). The programming models for use with real-time systems typically define: (i) a given set
of concurrency constructs; (ii) a range of communication and synchronisation means; (iii) a given notion of time; and (iv) a given interface with the external environment.

Our notion, however, is of a higher-level order than its correspondent. The notion of Computational Model, in fact, allows us to lift the concerns captured by the Programming Model to a higher level of abstraction, so that the real-time aspects of the system construction can be addressed at the design level also. This abstraction process, however, is only possible so long as the chosen implementation means conform to the abstract model upon which the static analysis is based. Hence, any proper instantiation of our notion of Computational Model demands the provision of a Programming Model that excludes from the "legal" implementation all those parts of the programming interface which may undermine the assumptions against which the system is verified. The notion of Computational Model was designed to capture these concerns. In the following we provide a more stringent definition of the concept of Computational Model and discuss alternative implementations which may suit our needs.

### 3.4.4 Conceptual Definition of Computational Model

A Computational Model provides the reference framework within which a real-time software system is designed, analysed and implemented; in particular, it allows the real-time behaviour of the system to be predicted, analysed and managed in a consistent manner.

As we have seen above, the Computational Model is an abstract notion which defines:

- the *components* from which the software system will be constructed, the *execution properties* and *real-time attributes* which characterise those components, and the *design rules or guidelines* for a consistent use of those components;

- the approach taken to and the means of *communication and synchronisation* among components as well as between those and the external environment;

- the *real-time execution model* assumed for the run-time execution of the system, expressed in terms of: the *scheduling paradigm* under which the system is to operate; the associated *scheduling mechanisms*; and the *analytical devices* which permit the static analysis of the real-time behaviour of systems thus constructed and executed.

Differing Computational Models can be classified along the following axes:

1. nature of the components used in the design; for example, in HOOD [HTG, 1993], the design components are *objects* offering constrained or unconstrained operations for other objects to invoke; these objects may be active and concurrent (and, hence, possess an independent thread of control) or passive (and possess none);

2. number of processors and nature of coupling (this dimension may also include the failure behaviour of the processing units so that a physical processor can be, for example, abstracted to the Fault-Tolerant Unit of the MARS system [Kopetz et al., 1989]);
3. *sequential* or *concurrent* execution and associated scheduling policy (which can be *static* or *dynamic*); examples of sequential execution model are cyclic and event-driven scheduling, whereas instances of concurrent model are cooperative and preemptive scheduling [Burns, 1991];

4. type of interactions with the external environment (interrupt-sensitive, deferred interrupts such as polling) and means of communication and synchronisation (synchronous, asynchronous).

### 3.4.5 Practical Instances of Computational Model for On-Board Systems

Several research projects have, over the last decade, proposed various approaches to the construction of predictable real-time systems. These approaches employ differing forms and combinations of hardware and software architectural support to the definition of the system. The extent of specialisation of the required hardware support varies with the criticality of the fault tolerance requirements placed on the system and the feasibility of implementation in the reference application domain. As the issue of fault tolerance falls outside the scope of our work, however, in the following we will not pursue the hardware aspects of this discussion any further and exclusively concentrate on the software aspects.

In general, the designers of "model" hard real-time systems are typically inspired by one of two contrasting goals [Strigini, 1993], which place on top of the guarantee of timeliness for critical (hard) tasks: (i) ease of verification; or (ii) efficiency of resource utilisation and adaptivity to variable load conditions. Emphasis on the former goal leads towards a *static* design approach which tightly constrains the possible execution histories of the system so as to better guarantee predictable behaviour. Emphasis on efficiency leads to systems capable of more *flexible* and *dynamic* behaviour.

MARS [Kopetz et al., 1989] is one of the most rigorous (and successful) representatives of the static approach, in addition to featuring very strong connotations of fault tolerance. Conversely, the Spring Kernel [Stankovic and Ramamrithan, 1989] may be regarded as a champion of the dynamic approach, yet removed from any explicit consideration of fault tolerance.

In MARS, all the tasks in the system and all communications between them are scheduled *off line* and the predictability of the system is fully and completely guaranteed at the cost of a rigidly defined and enforced design of the schedule. In the Spring Kernel, instead, the critical tasks (conceivably covering a definite subset of the intended operation of the system) are scheduled and guaranteed off line, while all other tasks are allowed to move about the system *at run time* until they find a processing node with enough spare capacity to guarantee their execution.

A variety of other systems can be added to this list and ranked at various points of the spectrum; for example, ARTS [Tokuda and Mercer, 1989], which ensures the timeliness of operation of the system under all conditions prior to execution and which we include in the *static* category; RT-MACH [Rajkumar et al., 1998], which provides timely and protected access to processing resources using a combination of resource reservation and admission control mechanisms and which we still consider as a borderline member of the *static* category; Alpha [Jenson et al., 1989], which
aims to guarantee the execution of the tasks with the highest value to the mission of the system at a given time and which we place in the dynamic category.

Next to this classification, [Kopetz, 1991] distinguishes between time triggered (TT) and event triggered (ET) real-time systems and draws rather drastic conclusions as to the greater suitability of the former for time-critical systems. In TT systems, the execution of the processing components is based on the regular progression of a common, coherent and guaranteed notion of real-time. Conversely, in ET systems, the execution of the processing components progresses as a result of the occurrence of events which may be internal (e.g.: a communication) or external (e.g.: an interrupt) to the system. Consequently, continues [Kopetz, 1994], the real-time kernels supporting the implementation of TT systems can be kept to a safe and efficient minimum (only having to cater for the management of task switching and global time). In contrast, the real-time kernels which support ET systems are demand-driven, hence intrinsically exposed to the temporal variability induced by the management of unpredictable activities (such as for interrupts and communications).

It is important to observe that all of the "model" systems we have considered so far have two distinct characteristics in common: (1) their Computational Model assumes some degree of physical distribution; and (2) their instantiation requires the use of a specialised real-time kernel. Both assumptions place these models clearly outside the confines of our solution space. We known from requirement R.2 in section 1.3, as well as from section 2.3, in fact, that the on-board embedded real-time systems addressed by this work are based on single-processor architectures, whilst the move to new-generation systems will only reinforce this trend. Hence, for as much as we may concur with [Kopetz, 1994] that distribution (and fault tolerance) are implementation characteristics that should not matter to the user, for timeliness is the service attribute which matters, we take here the view of the developer, for whom distribution is a most crucial discriminant. This simple fact excludes from our horizon all Computational Models which explicitly require physical distribution of resources.

We also know from the earlier discussion in this chapter that our challenge is to build upon the implementation technology presently in use at European space industry. This basically means that our main interest is for a Computational Model, the implementation of which can be effectively operated in an Ada environment and, ideally, with a degree of efficiency and predictability comparable with that of the MARS kernel [Reisinger, 1994].

It descends from the discussion so far that the Computational Models of practical relevance to our work constitute a tiny proportion of the whole range of options captured by the above classifications. In essence, within the rigid confines of our solution space, the emerging requirements will oppose the determinism and rigidity of off-line, non-preemptive scheduling to the extent of predictability and flexibility guaranteed by on-line, preemptive scheduling. The former would configure as a rigid implementation of the static, time-triggered approach; the latter as a contamination of the static approach with a degree of event-triggeredness.

In the following, we briefly review the foundations of two instances of scheduling and execution paradigms which form the basis of suitable yet alternate Computational Models for on-board embedded real-time systems. The review draws extensively from [Baker and Shaw, 1989], [Locke, 1992] and [Audsley et al., 1995].
Fixed Cyclic Scheduling. Cyclic scheduling systems are sequential (i.e. single-threaded) systems comprised of a fixed (off-line determined) schedule of non-preemptive actions or operations which are explicitly invoked at regular intervals of time by a run-time scheduling executive.

The schedule of calls invoked by the executive is statically established off-line. (In fact, as many fixed schedules must be provided as the different operation modes required of the system.)

The basic time period between successive executions of the entire schedule of calls is called a major cycle. It is often the case that some actions in the schedule must execute at a higher frequency than others. In this event, the action list in the schedule must include as many calls to those actions as required to meet their frequency requirement. Every single occurrence of such a call is called a minor cycle. The major cycle is thus decomposed into a sequence of minor cycles and its frequency is set to the least common multiple of all the required frequencies.

The internal progress of execution within the cyclic schedule must always be correlated to some external source of reference signal; typically a periodic hardware interrupt generated off an external clock. This signal is often referred to as the Real-Time Clock, RTC.

The need to maintain a simple correlation to this signal requires that the period of every minor cycle be set to the same value and be synchronised with the periodic generation of the RTC. (Accordingly, actions with the same – adjusted – frequency appear in the same minor cycle in some arbitrary order.) This approach requires that the frequency of the RTC must be equal to, or greater than, the frequency of the highest-frequency action. It follows that every action in the schedule must operate at integer multiples of the frequency of the RTC. This creates the artificial requirement for all actions to share a harmonic relationship.

The establishment of the schedule list (otherwise called timeline) requires the knowledge about the worst-case execution time of every action. The total execution time of all the actions scheduled within one (minor or major) cycle cannot exceed the assigned duration of the cycle. Actions which do not fit within the assigned cycle have to be split (by re-coding or re-design) into smaller actions and allocated to subsequent cycles. This corrective measure basically corresponds to hand-crafting the preemption process of those actions.

In a cyclic scheduling system, each action is normally responsible for completing its operation within the assigned time limit and for relinquishing control to the cyclic executive so that this can schedule the next action at the correct point in time. Failure to do so determines the occurrence of a cycle overrun. This phenomenon is observed when the arrival of the RTC occurs when an action is still in progress; however, as a minor cycle may be comprised of several actions, the occurrence of a cycle overrun can only be observed at the boundary to the next cycle and, hence, possibly just reflect the late effect of an earlier overrun in the same minor cycle. On this event, the cyclic executive may choose to allow the current action to continue and therefore delay the next due cycle; this may obviously perturb the execution of the rest of the timeline. Alternatively, the executive may abort the overrunning action; this preserves the rest of the timeline but exposes the system to the risk of creating inconsistencies in the persistent data of the application.

With respect to the classification criteria introduced earlier in this section, a Computational Model adhering with this paradigm will provide for:

1. a "world of objects" comprised of one active (threaded) object representing the cyclic executive; and any number of passive objects representing the individual actions in the timeline;
2. a single-processor environment;

3. a single-threaded non-preemptive scheduling and execution regime in which the active object sequentially invokes the operations provided by passive objects according to a pre-defined action list;

4. no internal synchronisation between passive objects; and polling for external interrupts.

**Deadline Monotonic Fixed Priority Preemptive Scheduling.** Systems which adopt fixed priority preemptive scheduling are *concurrent* (i.e. multi-threaded) and based upon a run-time environment in which tasks possess a fixed priority attribute and are preemptively dispatched in priority order. Tasks in such systems are either runnable, in which case they are held on a notional priority-ordered run queue; delayed, in which case they are held on a notional time-ordered delay queue; or suspended, in which case they are awaiting an event which may be triggered externally (via an interrupt) or internally (from some other task).

The response time of a task run in this fashion critically depends on the interference incurred from preemptive execution of higher-priority tasks. The fact that the priority assignment to tasks may be arbitrary complicates the analysis of their real-time behaviour.

The foundation of the analytical model for fixed priority preemptive systems were laid down by Liu and Layland in 1973 [Liu and Layland, 1973]. Their model, which came to be known under the name of *Rate-Monotonic Priority Assignment Policy*, provided a feasibility test for task sets comprised of any number of periodic and independent tasks with fixed computation time, deadline equal to period and fixed priority assigned in rate-monotonic order (the shorter the period, the higher the priority).

That feasibility test was sufficient but not necessary and the associated Computational Model was too restrictive for field applications. A great deal of research work went into defining models capable of relaxing some of the original restrictions (e.g.; task independence) as well as into the determination of the *worst-case response time* of tasks in these models. The feasibility test would in this case be trivially obtained by comparison of the value thus determined with the task deadline. (The reader is referred to [Audsley et al., 1995] for an historical perspective on such developments.)

In the deadline monotonic variant of the original rate monotonic scheme [Audsley et al., 1991, Audsley et al., 1993], the number of tasks in the system is fixed. Tasks in this system are library-level (i.e. with no internal hierarchy) and infinite; their regime may be either periodic or sporadic. In the former case, tasks consist of an infinite number of time-triggered invocation requests, each separated by a minimum amount of time which correspond to the task period. In the latter case, the invocation requests are event-triggered and separated by an amount of time which may vary between the minimum and the maximum interarrival time of the triggering event. The triggering event may originate from another task in the system or the environment (e.g.: an external interrupt).

Task deadlines may be shorter than their period or minimum interarrival time. Task priorities are thus assigned in deadline monotonic order.
Tasks are either totally independent of each other or allowed to share data which are kept in protected data structures. A protected data structure can be held by a single task at any one time. Tasks which require access to a protected data structure currently held by another task are blocked until that task relinquishes the lock on the shared protected data structure. The basic locking mechanism is exposed to priority inversion, which occurs as a result of a higher-priority task being blocked on access to a shared protected data structure held by a lower-priority task.

As described by [Goodenough and Sha, 1988, Sha et al., 1990], this adverse phenomenon is circumvented by having a temporary ceiling priority assigned to tasks on access to the protected data structure.

The ceiling priority is the unique attribute of every individual protected data structure and is set to at least one level higher than the maximum priority of the tasks which share the structure. The priority of tasks reverts to the initial level (i.e. the base priority) on exit from the protected data structure. The use of ceiling priority bounds the effect of priority inversion and ensures that it can be minimised by design. The relevant amount of blocking time can be easily determined by static analysis of the system.

The analytical model for the determination of the worst-case response time of tasks in this kind of system are discussed in detail in section 4.5.3.

With respect to the classification criteria introduced earlier in this section, a Computational Model adhering with this paradigm will provide for:

1. a "world of objects" comprised of any number of active (threaded) objects conforming with the periodic (time-triggered) task or the sporadic (event-triggered) task model; and any number of protected objects representing the protected data structures in the system;

2. a single-processor environment;

3. a multi-threaded preemptive priority-based scheduling and execution regime;

4. internal synchronisation between active objects via protected objects as well as with the environment in the form of external triggering events.

3.4.6 Our Choice

It is one key lemma of our work that there are important advantages to be gained with the adoption of a Computational Model based on fixed priority preemptive scheduling and directly implemented upon the revised tasking model of Ada 95 [ISO, 1995]. We insist that this choice serves our goals well in several respects: (i) it suits the emerging requirements of modern satellite systems discussed in chapter 1 and the inherent characteristics of on-board embedded software highlighted in chapter 2; (ii) it ensures the degree of implementation flexibility which we need to support our iterative and incremental development approach; (iii) it combines the guarantee of predictability with high efficiency of execution. We will discuss and justify this lemma further in chapter 4.
3.5 Relation to Other Practices

With the financial decline and fragmentation of the former Soviet Union, Europe has risen as the second-largest world contributor to unmanned space activities; the largest being the United States. It may, therefore, be of interest to position our review of the software practices advocated by the European Space Agency with respect to those endorsed by the Agency’s correspondent in the United States, the National Aeronautics and Space Administration, NASA. We do so in the following, looking into the process as well as the associated technology aspects.

3.5.1 Process Aspects

The software development life cycle model assumed by the PSS-05 standard places in almost perfect match with the model described in [SEL, 1992] by the Software Engineering Laboratory (SEL) for use in space developments funded by NASA. The cross-correlation between the two models reveals, in fact, a large number of commonalities and a few distinct differentiations. The broad range of commonalities confirms the overall conceptual validity of the PSS-05 model. The few differences denote slight, yet interesting, shifts of emphasis between the two approaches. Let us now take a quick look at both angles.

The software development life cycle as seen by SEL is as sequential as the PSS-05 model and comprised of 8 main phases with different names from but virtually identical scope to the corresponding PSS-05 phases. Notably, the SEL model requires the Requirements Analysis phase (equivalent to the SR segment in the PSS-05) to deliver a software specification with basically the same function as the logical model of the PSS-05. Similarly, the Preliminary Design phase of the SEL model (equivalent to the AD phase in PSS-05 terms) is to deliver a preliminary design report which can be regarded as an early version of the physical model required by the PSS-05. Beside these aspects, the SEL model effectively progresses across the same conceptual steps and notions as assumed by the PSS-05.

The most interesting differences between the two models arise in the respect of: (i) the greater emphasis laid by SEL on the Implementation phase, which, in fact, spans across the DD–IT segment of the PSS-05 model, explicitly anticipates a number of internal iterations — called "builds" —, and, accordingly, accounts for the largest proportion of the development effort; and (ii) the explicit acknowledgement given to the fact that, irrespective of the sequential nature of the development cycle, activities of earlier development phases tend to noticeably spill over the "formal" confines of later development phases.

Both observations touch on aspects of great interest to our work. Observation (i) denotes the greater attention attributed by SEL to the lessons learned from empirical and experimental evidence than apparent from the PSS-05 philosophy. On the whole, however, the observation also confirms our finding that the ascending branch of the V model actually absorbs the largest proportion of the development effort. We discussed the PSS-05 related side of this aspect in section 3.3.

In effect, the figures supplied by [SEL, 1996] for a wide range of NASA developments indicate an effort distribution rate of 26% for the SR–AD segment (against our 25-30%) 37% for the DD–UT segment (against our 15%) and another 37% for the IT–ST segment (against our 55-60%). The
different spread across the DD–ST segment reported by [SEL, 1996] clearly reflects the greater emphasis placed on implementation and the comparatively small representation of the on-board embedded software component over the baseline considered by the study (merely 6% of the total).

Observation (ii), instead, captures one of the considerations which stand at the very basis of our project: that is, the evidence that the formal exit from the "internal" development phases (i.e. those remote from both contractual ends) rarely coincides with the actual completion of the relevant activities. The impact of the latter aspect on the PSS-05 development model in the face of the emerging requirements was in fact one of the key concerns addressed by section 3.4.2.

3.5.2 Technology Aspects

Earlier in section 3.4.1 we already commented upon the experimental evidence of the benefit of using Ada and object-oriented design techniques reported by SEL in [SEL, 1993] and [Waligora et al., 1997]. We used that evidence in support of our choice to build, in an evolutionary fashion, upon the present European technology baseline centred around Ada [ISO, 1987] and HOOD [HTG, 1993].

In spite of the cited evidence, however, [SEL, 1992] takes a distinctly non-prescriptive approach to the selection of the reference enabling technology. There appear to be two main reasons for this choice. The first reason has to do with the negative reaction fostered by the imposition of engineering tools and practices on the workforce. This phenomenon is clearly described by [Waligora et al., 1997] as one of the primary causes for the decline in the use of Ada recently observed at NASA after the forced introduction of the language in the mid 80s. The second reason merely reflects the wide range of technologies required to accomplish the maintenance of the software systems in operation, along with those under development. In fact, in spite of the dramatic increase with the use of C, C++ and Ada for the more recent developments (which globally account for about 60% of the 166 million source lines scrutinised by [SEL, 1996]) the most part of the systems presently in operation were developed using a fairly large range of languages and tools.

More recently, under the effect of the same "cheaper, faster, better" paradigm discussed in our work, the emphasis at NASA has shifted on the wide-spread reduction of software costs, which [SEL, 1996] estimates to 8% of the total expenditures and 12% of the overall workforce dedication. This trend has resulted in the increasing prevalence of C and C++ over Ada on account of the still significantly higher procurement cost of the latter, as well as in the strong push for the increased use of commercial off-the-shelf (COTS) components. Recent space missions have not escaped this push and their on-board embedded real-time applications were thus built upon COTS operating systems. These experiences, however, have delivered mixed results (cf., for example, [Reeves, 1998] over the ultimate practicality and convenience of COTS for on-board applications).

Our position in the respect of the use of COTS is to side with [SEL, 1996] on the preference for the implementation of short-paced improvements to the standard process (what we have called the "evolutionary" approach) which the intrinsic volatility of COTS can hardly support.
3.6 Summary of Discussion

This chapter has maintained that, in the development of on-board embedded real-time systems, an important proportion of real-time requirements arise as second-order requirements along with the progressive establishment of the physical model of the system.

In contrast with this view, the PSS-05 standard [ESA, 1991] insists that all requirements on the system should be completely defined on exit from the software requirements definition phase. The definition of one feasible physical model which meets such requirements then becomes the exclusive object of the architectural design definition phase.

This chapter has argued that, in order for these contrasting views to be reconciled and for the logic, consistency and integrity of the overall development process to be preserved, the designer must be able to perform — in an iterative manner — an incremental verification of the feasibility (and optimality) of the logical and physical model of the system prior to finally submitting them to the activities in the ascending branch.

We have also shown that, whereas the systems of interest to our work are all distinctly concurrent and increasingly time-critical, the technology commonly used for the software design phase does not support any sufficiently expressive nor prescriptive notion of concurrency. We have thus argued that this deficiency impairs the validity and stability of resulting physical model. We have then presented the strategy which we propose to employ to overcome this problem. Our concept revolves around the explicit incorporation of the notion of Computational Model as the new centre of gravity of the physical design process. This notion allows the real-time aspects of the system to be addressed at the design level and provides a prescriptive definition of the relevant implementation requirements. We have argued that our provisions will augment the productivity of the development phase and present the verification phase with a system concept requiring a significantly lesser extent of testing of the real-time behaviour.

Chapter 4 will argue that the current software process does rarely perform any informed and justified selection of an appropriate Computational Model. Section 3.4.5 contrasted the traditional scheduling model (i.e. fixed cyclic scheduling) with the deadline monotonic fixed priority preemptive scheduling paradigm. Section 3.4.6 subsequently stated our preference for a Computational Model built on the latter form of scheduling, for that better suits our strategic objectives with respect to the optimum use of Ada and the preservation of the required degree of design flexibility. Chapter 4 will show that we can in fact meet the emerging requirements using technology which is directly supported by the Ada 95 language standard [ISO, 1995], along with powerful analytical models and tools to assist and assure the design and the implementation.
Chapter 4

An Evolutionary Approach to the Construction of New-Generation Systems

4.1 Introduction

This chapter combines the components of the engineering strategy outlined in chapter 3 into a comprehensive approach to the development of new-generation on-board embedded real-time software. In the remainder of this work, chapter 5 and 6 will demonstrate the operation and fitness for purpose of our concept.

The discussion in this chapter covers an extensive amount of material, which spans from the conceptual description of the proposed development model up to the definition, implementation and characterisation of the associated enabling technology.

The structure of this chapter is as follows:

a. section 4.2 introduces our revisited interpretation of the PSS-05 development model, illustrates its focus on the support for the iterative and incremental consolidation of the real-time structure of the system (i.e. the physical model) relates it to other known and relevant innovative approaches and summarises its overall rationale;

b. section 4.3 describes the structure and organisation of development activities in the process and identifies the requirements on the associated enabling technology;

c. section 4.4 provides an overview of the design method which we have adopted as the centre of our engineering approach;

d. section 4.5 presents the technology that supports our iterative and incremental approach to design, implementation and static analysis of new-generation systems;
e. section 4.6, finally, relates the projected benefits of our approach to the known limitations of current practice.

4.2 Methodological Approach

4.2.1 Outline of the Proposed Approach

Earlier in this work, we have seen that new-generation satellite systems will be increasingly software-intensive and confronted with the urge to dramatically compress the associated development schedule. With software becoming central to the implementation of such systems, the suitability of the current software process and associated support technology needs to be reconsidered in the light of the emerging requirements. This was in fact the main object of chapter 3.

Section 1.4 had anticipated the strategic line of our work and argued that the future software development process ought to be turned, from the classical waterfall model, into an explicitly iterative and incremental process. This is an essential component of the productivity boost required to cope with increased software complexity in the context of a dramatically compressed development schedule. The value of this strategic component was in fact advocated by chapter 3.

In anticipation of the increasingly concurrent and time-critical component of new-generation systems, section 1.3 had also called for a more effective development paradigm capable of mitigating the labour intensiveness of the real-time verification activities. Chapter 3 placed special emphasis on this particular aspect and outlined the essential ingredients of the desired evolutionary solution. In particular, chapter 3 highlighted one major deficiency of the PSS-05 development model [ESA, 1991] in the respect of time-critical systems. In fact, the PSS-05 model, which currently informs the software development practice of European space industry, fails to recognise that an important proportion of the real-time requirements on on-board software systems arise as second-order requirements in conjunction with the establishment of the physical model of the system. The PSS-05 model requires that the requirements capture feed into but be kept separate from the establishment of the physical model. This approach is therefore exposed to the risk that second-order requirements may escape the elaboration and analysis destined instead to all the first-order requirements captured in the designated phase.

The lack of explicit support for the capture and analysis of such an important baggage of requirements implies that no comprehensive verification of the system concept can effectively be carried out in the descending branch of the V-shaped PSS-05 development model (i.e. the design and implementation phase). This deficiency causes the entire burden of verification to be deferred to the ascending branch (the testing phase) only to result in the transfer of an improper extra load to the IT-ST segment of the process shown in figure 3.1. Section 3.3 discussed this phenomenon and nicknamed it the "snow-plough" syndrome.

The effects of the snow-plough syndrome are particularly antagonistic to the required reduction of the development schedule. Any significant result in that respect, in fact, cannot be obtained other than by decreasing the effort and complexity demanded by the testing activities, which often take up to 60% of the overall development effort. In our interpretation, this essentially means that the IT-ST segment must be offloaded of concerns which can be resolved earlier in the process.
Undoubtedly, in the face of the required reduction of the development schedule, the level of effort presently devoted to the testing phase cannot increase proportionally with the anticipated increase of the real-time complexity of new-generation systems. In view of this, section 1.4 and 3.4.2 have proposed to replace as much *late* dynamic real-time testing as possible by a functionally equivalent amount of static real-time analysis performed *earlier* in the process, as a remedy to the present ill-balancedness of the effort distribution across the software life cycle.

As discussed in section 3.4.3, we want to achieve this objective by introducing two distinct enhancements to the current software process, namely:

1. the notion of *Computational Model*, as the means to augment the expressive power of the design definition phase to capture: (a) the *real-time attributes and execution characteristics* of the components from which the software system will be constructed; (b) the *real-time execution model* which underpins the design of the system; and (c) the *means of communication and synchronisation* applicable among components and between them and the external environment;

2. the introduction of a unified *design framework*, embracing the whole spectrum of the real-time design activities (i.e. requirements analysis; design definition; design analysis and implementation) into a single, unfractured process stage within which the development can proceed in an *iterative* and *incremental* fashion and ordiately respond to the *feedback* arising from the progressive consolidation of the system.

![Figure 4.1: The Proposed Software Process.](image-url)
Figure 4.1 illustrates our development model and contrasts it with the conventional waterfall approach.

The figure shows: (a) the *design framework* (the dashed box tagged with a (2) in the figure) which embraces the feedback-based development iterations occurring within the the SR-DD segment of the V model along with the consolidation of the system concept; and (b) the notion of *Computational Model* (tagged with a (1) in the figure) which supports the establishment of the physical model, the adoption of conforming design rules and implementation choices (denoted by the left-headed arrows tagged "enforcement" in the figure) and successive and incremental stages of static timing analysis (denoted by the left-headed arrows tagged "verification" in the figure).

As the figure portrays, our process concept is geared towards the capture and accommodation of the design and implementation iterations caused by the feedback from requirements consolidation and/or timing analysis. The *design framework* is the structured development space in which such feedback can be analysed and responded to by design and/or implementation refinements. The *Computational Model* is instrumental to this objective, for it allows the real-time issues to be captured at the design level, coherently transformed into a conforming implementation and statically analysed at all stages of development.

Section 4.3 will build on this initial concept in the definition of the process model that this work wants to promote.

### 4.2.2 Relation to Other Development Approaches

It is commonly experienced that the description of a software architecture is not necessarily well served by a single architectural style. No single style, in fact, is normally capable of providing the "right" type and amount of information to all members and functions of one development team. In fact, there may exist as many styles of expression as the viewpoints that can be taken with respect to the software architecture as a product. Kruchten [Kruchten, 1995] voices this evidence and proposes a model which contemplates four concurrent views to describe the same software architecture and one common space in which the various views can come together through the illustration of selected use cases or scenarios.

[Kruchten, 1995] recognises the co-existence of: the *logical view*, to describe the functional behaviour of the system; the *process view*, to describe the concurrency and synchronisation aspects of the system operation; the *physical view*, to describe the topology of the system architecture; and the *development view*, to describe the static organisation of the system components in the development environment. Kruchten allows each view to have its own notation and be expressed in terms of the set of architectural elements, representations and associated rationale and constraints that suit the development perspective. [Kruchten, 1995] also advocates an *iterative* development in which the architecture is prototyped, tested, measured, analysed and refined in subsequent iterations. Kruchten claims that this is better suited for ambitious and unprecedented projects – for which too little is known at the end of individual phases to validate the architecture – than the conventional linear approaches which contemplate backtracking merely as an exception.

Admittedly, our approach has a lot in common with the views expressed by Kruchten. Especially as regards the aim to acknowledge and accommodate the occurrence of explicit develop-
ment iterations as well as the need to use specific means (which [Kruchten, 1995] calls scenarios and we call static analysis) to progressively refine and consolidate the system concept.

Of the four views advocated by Kruchten, we are not interested here in the physical and the development views, but we have clear parallels between his logical view and the PSS-05 logical model and his process view and the PSS-05 physical model. In fact, we have discussed at length in chapter 3 our criticisms to the PSS-05 model for not being able to maintain the integrity of the system concept across the development of the logical and the physical model. In there, we have also argued that the basic HOOD architectural style is insufficiently equipped to express and master the real-time aspects of the design. In order to cure these deficiencies we have introduced: (i) the design framework, as the common space within which the two dimensions of the process can progress coherently and concurrently; and (ii) the Computational Model, as the key structural ingredient of the real-time architecture of the system.

Our next step will be to show that the architectural notation used to describe the physical model can be based on the same, yet enhanced conceptual framework used for the logical model; which provision, in our opinion, greatly simplifies the iterations between the two.

On top of this, we must also note that the basic rationale of our development approach apparently originates from the same motivations which underpin the design methodology advocated by Fohler and the MARS team in [Fohler et al., 1990]. The goal of [Fohler et al., 1990] is to propose a design methodology (seen as a collection of complementary development methods and the relevant use rules) specifically suited for the construction of time-critical systems. [Fohler et al., 1990] argues that the construction of real-time application software which does not consider the characteristics of the target system is totally inadequate. Moreover, it insists that the consideration of time must form an integral part of the design methodology, from the initial requirements definition down to the implementation level. [Fohler et al., 1990] maintains that time must be represented in a uniform and concise way all along the development process. Finally, [Fohler et al., 1990] advocates the need for means capable of supporting the early evaluation of the design with respect to its timeliness requirements. Although basically identical as to the motivations, our work departs from that of Fohler et al. in the respect of technical choices made to respond to the needs. (As a reflection of this, chapter 3 contrasted our approach to the philosophy of MARS.)

4.2.3 Rationale of the Proposed Approach

The operational correctness of real-time systems needs to be verified in the value domain as well as in the time domain. The latter concern demands that attention be paid to the constraints and properties of execution on the run-time environment which may effect the time-domain behaviour of the system (e.g.: hardware architecture, clock frequency, means of communication with the external world).

For embedded real-time systems in general and on-board systems in particular, the nature of the execution environment not only effects the time-domain behaviour of the system but also plays a crucial role in the determination of its structure and operation. The most part of modern real-time systems use run-time executives or operating system kernels to hide the properties and
constraints of execution on the run-time environment into an abstract "presentation layer" offered to the application components.

As a reflection of this basic engineering principle, real-time systems are normally structured in the classical layered fashion. Consequently, the application components execute on top of a run-time executive and are designed around the services, means and abstractions that the executive provides.

Hence, the structure of applications in this domain is determined, to a great extent, by the need to fit the abstractions and execution model supported by the run-time executive of choice.

This need obviously spawns a set of additional requirements on the transformation of the logical model of the system, as seen by the PSS-05 model discussed in chapter 3, into the physical model submitted for implementation. In order to denote their derivative origin, these additional requirements are normally referred to as second-order requirements.

As opposed to first-order requirements, which directly originate from interpretation of the user specification, second-order requirements emanate from design decisions which are autonomously made by the design authority. Irrespective of their differing origin, though, both complements of requirements equally determine the design of the system and therefore call for equally accurate verification and verification.

The above concern was captured by section 3.3.1 and formulated as follows:

**A.1** in the development of real-time embedded software, a considerable amount of second-order requirements is incrementally defined in conjunction with the progressive establishment of the physical model of the system.

Hence, the timely consolidation of all of the requirements which effect the design and operation of the system – which is central to the stability and cost-effectiveness of the overall development – critically depends upon the timely establishment of the physical model.

In response to this concern, section 3.3.2 has asserted that:

**A.2** the establishment of the physical model is greatly facilitated by the explicit and consistent utilisation of a suitable Computational Model as the means to structure and automate the verification of the operational correctness of the system in the time domain.

The concept of Computational Model was introduced in section 3.4.4 with the aim to capture and formalise, as explicit design drivers, the properties and constraints which determine the execution model of a real-time application.

Assertions A.1-A.2 are central concerns to the conception of the software process advocated by this work. We now want to refine the formulation of these assertions in a fashion which better reflects the logic of the process which ensues from them:

**A.1.1** the establishment of the physical model of the system proceeds in conjunction with the incremental capture, refinement and formalisation of the second-order requirements which complement the set of direct requirements established with the SR phase of the V model;

**A.1.2** the introduction of second-order requirements may give rise to important amendments and refinements to the design of the system as determined from the logical model;
A.1.3 the development process which ensues is inherently iterative and incremental as a reflection of the progressive consolidation and verification of the physical model and its impact on the original logical model;

A.2.1 the consistency and productivity of this process are greatly facilitated by the explicit selection and enforcement of a suitable Computational Model and the systematic verification of satisfaction of the required properties and constraints.

Hence, in order to achieve the research objectives set forth in section 1.4 in the light of the above assertions, we need to establish a software development process which:

C.1 make of selection, enforcement and instantiation of an appropriate Computational Model an explicit and integral part of the design process, so that definite rules and means for system verification in the time domain can be effectively applied well ahead of integration and system testing;

C.2 cater for automated, incremental and iterative verification of the feasibility in the time-domain of the current design since as early in the development as possible, so that every such step of verification contributes to offload this concern from the effort and complexity of integration testing;

C.3 provide support for immediate analysis of the effect of modifications, enhancements and adaptations to the system which may result from late consolidation of requirements, late occurrence of timing problems, need for in-flight modifications, etc., so that the development process may become more controllably iterative and incremental and therefore achieve the increased internal parallelism which is essential to compress its duration;

C.4 be efficiently implemented using feasible and practicable enhancements of the HOOD and Ada technology presently in use, so that its introduction can effectively be as smoothly evolutionary as prescribed by section 3.3.

Each of the above-stated properties originates from assertions, requirements and assumptions made at various points in the preceding chapters. In order to help the reader take an overall view of the mutual dependences between such arguments, table 4.1 traces requirements C.1-C.4 back to their originating motivation as presented in this thesis.

<table>
<thead>
<tr>
<th>Required Property</th>
<th>Justification</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>assertion A.2.1</td>
<td>sect. 1.4 &amp; sect. 3.3.1</td>
</tr>
<tr>
<td>C.2</td>
<td>assertion A.1.1-3</td>
<td>sect. 1.4 &amp; sect. 3.3.2</td>
</tr>
<tr>
<td>C.3</td>
<td>requirement R.1, R.3</td>
<td>sect. 1.4</td>
</tr>
<tr>
<td>C.4</td>
<td>strategic decision</td>
<td>sect. 3.3</td>
</tr>
</tbody>
</table>

This thesis maintains that one feasible instance of a development model capable of adequately satisfying such properties can indeed be instantiated and enacted with conventional, yet enhanced, PSS-05-based technology, in keeping with the required evolutionary approach. Yet, no instance of PSS-05-based development model known to the author of this thesis (e.g.: [ESA, 1992b], [ECSS, 1997] and other industry-own standards in Europe) appears to exhibit enough of the required properties.

The constituting elements of the proposed solution, which have been outlined in section 4.2.1, are discussed in detail in the following.

4.3 Supporting Concepts and Technology Requirements

4.3.1 Essential Elements of the Proposed Solution

The strategic decision discussed in section 3.3 requires that the proposed process amendments be introduced in an evolutionary fashion. We show in the following that this can be effectively accomplished by introducing a limited number of enhancements to the HOOD [HTG, 1993] and Ada [ISO, 1987] technology which presently constitute the pillars of the software technology in use at European space industry (cf. e.g.: [Ratcliffe, 1995]).

In particular, we contend that the methods and tools required to support the proposed development model can be constructed along the following directions:

a. the introduction of a few definite enhancements to the HOOD method, aimed to augment the conventional design process with the expressive power, constructive guidelines and verification means provided for by the Computational Model of choice;

b. the implementation of the selected Computational Model upon a restricted form of Ada 95 tasking [ISO, 1995], in accordance with the fixed priority preemptive scheduling model selected as the preferred model in section 3.4.6, in a fashion which:
   • suits the type of applications to be found in the reference domain (cf. chapter 2);
   • is directly supported by the recent revision of the language standard;

c. the definition and integration of the enabling technology for the support of: (1) the advanced verification of the feasibility in the time domain of the physical model of the system as well as (2) the effective deployment of feedback-based iterative design.

As discussed in section 4.2.1, item a. and c. above represent essential enhancements to the current software process. Enhancement b., on the other hand, is an arbitrary corollary motivated by the decision to retain Ada as the reference enabling technology.

The design framework shown in figure 4.1 is the core of the process model we envision. We now want to define the break-down structure which implements that process in an evolutionary fashion and determine the flow of activities that are to occur within it.

The development activities within our design framework revolve around the four main stages depicted in figure 4.2, as follows:
B.1 *real-time design*: this stage of the process is initially entered with the definition of the problem as determined by the system specification (i.e. the user requirements); this stage encompasses the definition of the logical and the physical model of the system and focuses on the characterisation of the real-time attributes (commitments) required of the individual components of the system; the activities at this level are supported by the use of an enhanced version of the HOOD method;

B.2 *binding to Computational Model*: this stage of the process is entered at all times the abstract design of the system is submitted to implementation; at this point, the application program corresponding to the current level of design is automatically extracted from the design tool and fed to the designated compilation system; the compilation system binds the program to the selected Computational Model (thereby ensuring the use of the appropriate programming interface) and determines the processing requirements associated with the worst-case execution time (WCET) profile of the application components;

B.3 *static analysis*: this stage of the process is entered to statically determine whether the current implementation of the system is capable of meeting the real-time commitments of the corresponding specification when executed on the designated target platform; at this level, we perform static response time analysis upon a stylised representation of the system and associated real-time attributes, as derived from the current design (B.3(a) in figure 4.2) and the WCET profile of the relevant program, as generated by the compilation system (B.3(b) in figure 4.2);

B.4 *feedback to design*: this stage of the process is entered to review the results obtained from
the earlier stage of static analysis and determine the corrective measures (if any) required to ensure that the system implementation meets the designated real-time requirements; the determinations established at this level are then fed back to the design level in order for the designer to keep current the actual real-time attributes of the system and either conclude the design process or perform further steps of design (and/or implementation) increment and corrective iteration.

The process model outlined above assumes a design method which supports the construction of a real-time system in terms of the abstractions, execution characteristics, usage rules and constraints defined by the Computational Model of choice (cf. B.1). For systems designed in this manner, the implementation effort is comprised of two complementary activities (cf. B.2): (a) the derivation of the *concurrent structure* of the system, which is obtained by transformation of the structural components of the design into the corresponding code structures, as required by the adopted implementation of the Computational Model; and (b) the incremental production (by hand-coding or other means) of the *functional components* of the system and their insertion in the relevant structural element. At any stage of this development process, the system thus constructed is comprised of: (i) the *real-time requirements* to be met by the system (cf. B.3(a)); (ii) the *execution profile* of the system corresponding to its current concurrent structure and the timing measurements or estimates for its functional components (cf. B.3(b)); and (iii) the *timing characteristics* of the execution environment. This information base is used to perform static analysis of the real-time feasibility of the current version of system. The results of this analysis are reviewed by the engineering authority. The relevant deliberations are then fed back to the design level either to leave the design process (and enter the subsequent phase of testing, as shown on the upper-left corner of figure 4.2.1) or to iterate over the corrective consolidation of the design and implementation (cf. B.4).

The implementation of each of these stages of development will be illustrated in chapter 5 by means of a simple demonstrative example. Section 4.3.2, in the following, instead discusses the technology requirements for the implementation of the proposed concept.

### 4.3.2 Technology Requirements

Section 4.2.1 has introduced the evolutionary amendments to the current software development practice which are required to support the process depicted in figure 4.2. In the following, the three main constituents of the proposal (identified as enhancement a.-c. in section 4.3.1) shall be presented and discussed in isolation.

**Selection and Enforcement of Computational Model**

Enhancement a. calls for *the introduction of definite enhancements to the HOOD method aimed to augment the conventional design process with the expressive power, constructive guidelines and verification means provided for by the Computational Model of choice;*

The use of the HOOD design method, as presently defined, does not require nor prescribe nor even support the selection, enforcement and verification of any given Computational Model.
A HOOD design, in fact, is *static* in that it establishes the functional and operational interfaces between objects without being equally prescriptive on the *dynamic* aspects of the execution of the system. The HOOD method only allows for a *loose expression of concurrency* (i.e. the "control view" required to describe the overall control structure of the system) which, on the contrary, ought to be regarded as one fundamental ingredient of the design of modern on-board systems.

In HOOD, basic rules exist for the designer to determine how threads of control are allocated to individual objects, but no binding prescriptions are formulated to dictate the rules governing their concurrent execution at run time. In fact, this lack was rather deliberate, for the designers of the HOOD method opted to trade prescriptiveness for generality.

Two components of the method may be used to express constraints on the execution model of the system: (1) the *protocol constraints*, which govern the interaction between the object requiring a given constrained service and the object providing it (cf. section 3.2.2 for a discussion of the basic protocols); and (2) the *object control structure*, which is the design structure intended for the user to express the constraints placed on the control behaviour of the object. The very existence of protocol constraints on an object’s operation determines the *active* status of that object. The definition of active object does in turn assume the possession of own threads of control. Hence, active objects are intended to execute concurrently. Yet, no other means than the two mentioned above are provided to express the desired properties of the concurrent execution of active objects.

Even in the absence of a rigorous definition and explicit support for the determination of the execution aspects, however, the design of an on-board real-time control system *always* eventually entails the coupling between the application components (i.e. the HOOD objects) and the execution platform. This necessity should take to the foreground the constructive elements of the Computational Model of choice. More frequently, however, this step is limited to mechanically forcing the needed run-time bindings into the application, with only late consideration given to the real-time properties of the concurrency model imported with the run-time executive of choice.

This phenomenon denotes two important deficiencies of the typical development approach: (1) the lack of effective means for the early enforcement and verification of the required real-time behaviour of the system, which is clearly not addressed as a concern of the design level; and (2) the divorce of the architectural design process from the concepts and abstractions which emanate from the execution platform.

These deficiencies may incur onerous consequences, especially at the critical boundary between the design and the coding phase. They may in fact often result in the loss or decay of such important properties as:

- *design integrity*
  when components, services and interfaces provided by the underlying run-time system (that is, the incarnation of the chosen Computational Model) are not designed, modelled nor verified with the same degree of accuracy or detail as demanded for the application software;

- *functional cohesion*
  when requirements arising from the selected Computational Model introduce system partitioning criteria which effect the structure of the system as established in the design phase; this is, for example, the case when the need to allocate system activities to fixed cyclic exe-
cution slots introduces a system breakdown structure which differs from the one committed at the end of the design phase (note that, with fixed cyclic scheduling, this case normally occurs as late as at integration testing, when the time-to-completion pressure is typically so high that the required structural changes to the system are treated *locally* at the code level without returning to the design level);

- **verification coherency**
  when the verification of the correct execution of the system on the adopted Computational Model is not performed as part of the design verification but is deferred until integration testing.

In our reference scenario, the core of the design process is carried out by use of the HOOD method. We therefore devise and introduce enhancements to the method which can instigate an *explicit* process of selection, enforcement and binding with the desired Computational Model.

We do maintain that the definition, implementation and enactment of the required enhancements is in fact possible and economically achievable. Section 4.3 presents the proposed solution.

**Predictable and Characterisable Concurrency**

Enhancement b. calls for the implementation of the selected Computational Model upon a restricted form of Ada 95 tasking [ISO, 1995] in a fashion which: (i) suits the type of applications described in chapter 2 and (ii) is directly supported by the recent revision of the language standard.

Chapter 1 has asserted that new-generation on-board systems are increasingly concurrent and time-critical. Chapter 2 has shown that the control activities to be performed by such systems exhibit a broad variety of execution requirements. Hence, the Computational Models for use in the construction of such systems should desirably support flexible and predictable forms of concurrency.

The decision to adopt an evolutionary approach to the amendment of the current software technology raises the obvious question as to how well Ada positions itself with respect to such demands. The question is particularly intriguing in that, in contrast to the vast majority of the most commonly used implementation languages (Fortran, C, C++) the definition of the Ada language embodies the elements of a preemptive priority-based (PPB) Computational Model.

The definition of the Ada 83 tasking model [ISO, 1987], however, was admittedly too broad and general and exhibited several important shortcomings (e.g.: bulky heavy-weight implementation, exposure to priority inversion problems) which made it utterly unfit for usage in space systems as well as in the vast majority of other high-integrity systems.

Because of such a distinct baggage of deficiencies, Ada 83 was solely employed as a plain procedural language provided with custom-made bindings to the run-time executive of choice.

The divorce from the standard run-time environment of the language and the strategic need for every system builder in the business to possess in house the elements for ready-made custom solutions gave rise to the birth of several differing implementations of run-time executives.
intended to support Ada-based real-time on-board systems. The short-sightedness of most commercial competitors, however, caused those endeavours not to join forces with major international standardisation initiatives, such as, for example, the Catalog of Interface Features and Options (CIFO) proposed by the Ada Run Time Environment Working Group under the auspices of the ACM [ARTEWG, 1991] and, to a lesser extent, [ExTRA, 1994]. Not surprisingly, those industrial efforts invariably resulted in proprietary implementations, which were local (i.e. non-standard across company boundaries) and semi-formal (i.e. hardly described in terms of any proper Computational Model).

One common characteristic of those proprietary developments was the generalised rejection of the Ada 83 run time system along with its tasking model. In spite of this, however, some restricted form of PPB scheduling built around custom run-time environments progressively started to appear and be used in an increasing number of space applications. The proprietary executives described in [Matra Marconi Space and GSI-Tecsi, 1990], [MBB, 1991] and [Saab Ericsson Space, 1996] are amongst the most notable examples of this trend. We read this as a reflection of the distinct demand for flexible ways to model the inherent concurrency of the on-board processing activities.

Building on this evidence and along the line of argument initiated by [Locke, 1992] and subsequently developed by [Burns and Wellings, 1995a], we argued in section 3.4.6 that compelling reasons now exist to re-consider the suitability of the amended Ada 95 tasking model [ISO, 1995] for new-generation on-board system. This choice would buy the designer the ability to build on-board embedded real-time applications directly upon an international standard instead of upon custom-made variants. In particular, section 3.4.6 maintained that a Computational Model based on the deadline monotonic fixed priority preemptive scheduling paradigm embodied in the Ada 95 run-time support suits well the emerging requirements of new-generation satellite systems discussed in chapter 1 and the inherent characteristics of platform applications highlighted in chapter 2.

Section 4.5 will take this argument further and will demonstrate that we can define, implement to maximum efficiency and fully characterise the Ada run-time support suited for the exploitation of our Computation Model.

Support for Feedback-Based Iterative Design

Enhancement c. calls for the definition and integration of the enabling technology for the support of: (1) the advanced verification of the feasibility in the time domain of the physical model of the system as well as (2) the effective deployment of feedback-based iterative design.

One distinctive property of the our approach stems from the aim to translate the design framework concept introduced in section 3.3.2, to a practicable paradigm of development, centred around the explicit recognition of the iterative and incremental nature of the design process and the ability to provide direct support for it.

The goal is, in this respect, to enable the need for the design iterations shown on the left-hand side of figure 4.1 to arise – and, to some extent, be promoted – under the assistance and guidance of support tools tightly integrated with the design process.
The central concept is, thus, for the designer to be able, *at several points of the development process* (and ideally, as early as during the establishment of the logical model) to obtain prediction, confirmation and verification or otherwise of the correct timing behaviour of the current physical model of the system. This would allow the definition and implementation of any necessary corrective actions to initiate well before the point in time when their impact and cost may become intractable.

It is obviously important for the overall economy of the development that such an iterative development concept place *within* the normal design process as opposed to *without* it; which, in a sense, blurs the separation which the PSS-05 standard introduces between the SR and AD phase, as shown by figure 4.3.

![Feedback-Based Iterative Design Flow](image)

**Figure 4.3: Feedback-Based Iterative Design Flow.**

The *realisation* of such a concept requires the ability to extract from the current design sufficient information for successive stages of (schedulability) analysis to be performed and continually feed back to the design process.

The nature of the information required to perform such an analysis is manifold and depends on the actual type of scheduling analysis which one wishes to adopt. In general, however, one portion of the required information may directly emanate from known requirements on the current design (e.g.: period, deadline, precedence activation constraints, functional and operational dependences such as those expressed by the "use" relationship in HOOD) whereas another portion shall necessarily require knowledge about the execution performance (represented by the relevant WCET profile) of the code presently or expectedly associated with the design.

Obviously, as the design evolves and consolidates, so do the stability and solidity of the extracted information. Hence, the need to enable this process to be iteratively repeated with the progress of the development requires the ability to progressively replace initial timing predictions.
with actual measurements based on the code being produced.

**Limiting Factors to Achievable Innovation**

It should be apparent to the reader that, in the face of the wealth of the possible technology advances which might arguably meet the demands discussed in chapter 1, this thesis has opted to set forth on a comparatively modest rate of innovation.

The reason for this resides with the need for any proposed technology innovation to negotiate with the non-technical factors specific of the industrial domain, which tend to set tight bounds to the achievable extent of innovation. And this is particularly the case of such industrial domains as the space industry in Europe, characterised — until the present moment —, as discussed in chapter 1, by very specific technical needs and rather modest margins of investment, which justify the option in favour of the evolutionary approach discussed in section 3.3.

Even within a moderately evolutionary scenario, however, non-technical considerations may play a ponderous role. It may, thus, be anticipated that the relative merit of the propositions put forward by this thesis shall be scrutinised by the intended user community not solely from a technical stand-point. This is likely to be the case with the requirement for enhancement b. in particular, which may constitute one of the main cultural obstacles to the general acceptance of the overall proposal presented in this work.

The anticipated issue has the following two facets: in recognition of the anticipated increase with the event-driven and concurrent nature of new-generation software-intensive satellite control systems, the formulation of enhancement b. maintains that the desired Computational Model for use in such systems should be based upon fixed priority preemptive scheduling and hence employ a restricted form of the Ada 95 tasking model.

This proposition, however, places in striking contrast with the commonly-held view that onboard control systems must be strictly deterministic in order to be predictable (where the term "deterministic" is used to denote a system in which the execution sequence of its components is completely predetermined and fixed for the entire operation of the system, as opposed to the potential non-determinism of fixed priority preemptive scheduling systems).

The contrast between these two forms of scheduling is dated and certainly not exclusive to the space domain. Once deprived of its deprecated "religious" connotations, though, this contrast has resolved, in numerous application domains (e.g.: avionics, car industry, process control) into a fairly straightforward pragmatic design decision, normally made in full awareness of the relative pros and cons of either option. As a result of which, an increasing number of moderate-criticality applications have successfully transitioned to the use of preemptive scheduling.

Papers like [Locke, 1992] have greatly contributed to clarify that, whereas real-time systems "must be capable of providing a provable prediction of the ability of their design to meet all of its timing requirements" — whence the term predictability —, determinism is, indeed, sufficient for predictability but not necessary to achieve it. And, as the body of the rate monotonic scheduling theory [Klein et al., 1993, Lehoczky et al., 1989] stands to demonstrate, provable predictions can be routinely obtained for preemptively scheduled systems.

The acquisition of such a glaring evidence, though, is not yet seemingly in sight for the space domain; quite possibly because of the notorious 8-year delay which apparently takes for ground
technology to reach and penetrate the space market (cf. [ESA, 1995b]).

It is clear, therefore, that the introduction of the proposed development model shall have to overcome not only the known technical difficulties associated with the use of fixed priority pre-emptive scheduling (e.g.: more demanding run-time environment and potential pessimism incurred with the worst-case scheduling analysis techniques required to prove the predictability of the system) but also the cultural barriers which continue to deny all forms of non-deterministic concurrency in general, and Ada tasking in particular, any possibility of utilisation in critical components of space systems.

4.4 HRT-HOOD as the Centre of the Development Process

The HRT-HOOD method [Burns and Wellings, 1994, Burns and Wellings, 1995b] was defined by the Real-Time Group at the University of York (England) in the frame of a study programme initiated by the European Space Agency with the aim to devise ways to improve the productivity of HOOD and Ada technology in the construction of new-generation real-time on-board systems. The author of this thesis actively participated in the project and the main lessons learned from the programme (which are summarised in [British Aerospace, 1993]) provided the main direction to the definition of the proposal discussed in this thesis.

It is not the intent of this thesis to provide an exhaustive presentation of the method. The interested reader is referred to the relevant literature, in particular [Burns and Wellings, 1994] and [Burns and Wellings, 1995b]. In the following, we shall only concentrate on the features which are specifically relevant to the operation of the proposed concept.

The central goal of the method was to amend the base HOOD method with all of the additions and constraints deemed necessary to ensure that the design can be statically analysed for its timing characteristics at all stages of development.

To this end, and in recognition of the requirements discussed in chapter 1 and 3, the definition of the HRT-HOOD method was geared towards the following objectives:

1. to promote evolutionary enhancements to the base HOOD method that would not break but retain the principles of the base method;

2. to establish explicit and direct bindings between the constructs, concepts and constraints of design and a definite instance of Computational Model based on the revised form of the Ada 95 tasking model;

3. to instigate the conduction of an iterative design process prompted by the feedback information obtained from static analysis of the real-time properties of the system.

In keeping with the mandated evolutionary approach, the HRT-HOOD method bases its root in HOOD [HTG, 1993] (the main features of which have been briefly presented in section 3.2.2) and enhances it along the following dimensions:

- recognition of the type (e.g.: periodic, sporadic, protected) and criticality (e.g.: hard, soft, non-critical) of the typical real-time control activities of the application domain
characterisation of the real-time requirements (e.g.: period, offset, deadline) set on such activities

- definition, utilisation and enforcement of those design devices (e.g.: object cooperation exclusively via protected data structures so as to ensure boundedness of blocking) which comply with the rules, properties and constraints set out by the chosen Computational Model for the sake of resulting in a statically analysable design.

Building on the programming abstractions supported by Ada 95 [ISO, 1995], the HRT-HOOD method retains the passive object exactly as defined in HOOD; amends the definition of the active object by differentiating between:

- cyclic objects, which denote active objects used to model periodic activities

- sporadic objects, which denote active objects used to model event-triggered sporadic activities;

and introduces the protected object to denote those objects which provide protected access to shared data structures. In line with the corresponding language abstraction, protected objects (likewise HOOD active objects) have control over when their invoked operations are executed, but (unlike active objects) need not have any independent thread of control.

While retaining all of the HOOD standard execution requests described in section 3.2.2, HRT-HOOD also supports the following two additional constrained operations on protected objects which ensure mutually exclusive access to the protected data required by the operation:

protected synchronous (PSER) : when control flow in the client is interrupted until completion of the protected operation; PSER operations can be state-constrained.

protected asynchronous (PAER) : when control flow in the client is interrupted only until the execution request has been received (along with any required input parameters) and acknowledged by the protected object; PAER operations cannot be state-constrained.

Protected objects can also provide any number of unconstrained operations.

Cyclic and sporadic objects may support execution requests for asynchronous transfer of control (ATC) which is intended to allow for other objects in the system to command the immediate termination of the current operation of the object. ATC operations are asynchronous only and are named ASATC for short.

Sporadic objects must also provide for the execution request which is to trigger their sporadic operation. This execution request is modelled by an unconstrained START operation. On the whole, therefore, cyclic terminal objects may solely and optionally provide for an ASATC operation, whereas sporadic terminal objects may optionally provide for an ASATC operation and compulsorily for a START operation.

HRT-HOOD disallows the "use" relationships which would cause objects to incur unbounded blocking or arbitrary synchronisation. Furthermore, in keeping with the hierarchical object-based nature of the HOOD method and in order to allow non-terminal active objects to be meaningfully
marked as cyclic, sporadic and protected, HRT-HOOD introduces some restrictions on the allowable parent-child relationships so that the type properties of the parent object are not violated by any of its child objects. The relevant prescriptions are shown in table 4.2 and 4.3.

Table 4.2: Disallowed "Implemented By" Relationships.

<table>
<thead>
<tr>
<th>CHILD</th>
<th>cyclic</th>
<th>sporadic</th>
<th>protected</th>
<th>passive</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cyclic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sporadic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>passive</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 shows that, while retaining the HOOD requirement that passive parent objects must solely decompose to other passive child objects, HRT-HOOD adds the corresponding prescription that protected parent objects cannot decompose to cyclic nor sporadic child objects as the properties of the latter are incompatible with that of the former.

Table 4.3: Allowed & Disallowed "Use" Relationships.

<table>
<thead>
<tr>
<th>USED USER</th>
<th>cyclic</th>
<th>sporadic</th>
<th>protected</th>
<th>passive</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sporadic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>ASATC</td>
<td>ASATC, START</td>
<td>unconstrained</td>
<td></td>
</tr>
<tr>
<td>passive</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3 shows that passive objects can only use the operations of other passive objects whereas protected objects can call operations of any other active (meaning cyclic, sporadic or protected) objects so long as they are unconstrained (which ensures that their execution time can be bounded).

On the whole, HRT-HOOD objects are described by: their provided operations; their threads of control; their synchronisation requirements; and the real-time requirements set on their operations (i.e. criticality, period – or interarrival time –, deadline, WCET and priority).

Criticality, period and deadline are user-assigned design attributes and are expected to change only as a result of a design change. Conversely, WCET and priority are computed properties; the former may vary with the progress of the development (i.e. from a best-guess prediction, made at a very early stage of design, to the value returned from WCET analysis against the final code); the latter is calculated by the schedulability analysis tool based on the user-assigned criticality and deadline of the object in question.

So long as the user maintains the design information current with the progress of the implementation, the semantic contents of an HRT-HOOD design is such that the associated analysis
tools permit to constantly verify its feasibility in the time domain and, accordingly, derive the feedback necessary to steer the design and implementation to completion.

Figure 4.4 depicts the instantiation of the abstract process model shown in figure 4.2 that we have constructed around HRT-HOOD.

![Diagram of HRT-HOOD process model](image)

Figure 4.4: Design Activities in Operation.

The figure places emphasis on the iterative nature of the development flow we want to support. In our concept, such iterations represent incremental stages of development supported by various forms of feedback information. Static analysis constitutes our primary source of feedback. The figure shows the main components of the feedback from static analysis, that is: (1) the confirmation of the (schedulability) of the individual threads of the system with respect to the relevant real-time requirements; and (2) the sensitivity of the system timeliness to variations in the processing requirements of individual threads.

Notably, the implementation of an HRT-HOOD design aid tool supporting all of the data paths shown in figure 4.4 has recently been completed [Intec Sistemi, 1996a] and successfully demonstrated. The dashed rectangle in the figure depicts the sole stage of the process which is not assisted by automated tools and relies upon the engineering authority’s interpretation of the results obtained from static analysis.

In the following, we describe the key technology elements which are required to enable the effective operation of our process model.
4.5 Enabling Technology

4.5.1 The HRT-HOOD Computational Model

HRT-HOOD [Burns and Wellings, 1994, Burns and Wellings, 1995b] extends the base HOOD method by incorporating the abstractions supported by the revised tasking model of Ada 95 [ISO, 1995]. As a reflection of that, the HRT-HOOD Computational Model is based around the principles of fixed priority preemptive scheduling described earlier in section 3.4.5. Moreover, our implementation of the HRT-HOOD Computational Model places in perfect match with the tasking profile identified at the 8th International Real-Time Ada Workshop as the most suitable candidate for the support of concurrent time-critical applications [Baker and Vardanega, 1997] and referred to as the 'Ravenscar profile'.

Systems designed with the HRT-HOOD method are concurrent and provided with a direct mapping to the corresponding concurrency constructs at the language level. In particular, as outlined in section 4.4, the HRT-HOOD Computational Model:

1. retains the passive object exactly as defined in HOOD (i.e. an entity which has no internal concurrency and no control over when its invoked operations are executed);

2. amends the HOOD definition of active object (corresponding to an entity with internal concurrency and control over when its invoked operations are executed) by differentiating between:
   - cyclic objects, which denote active objects used to model time-triggered activities; and
   - sporadic objects, which denote active objects used to model event-triggered activities;

and, finally,

3. introduces the protected object to denote those objects which provide protected access to shared data structures.

HRT-HOOD active objects are threaded and possess a well-defined mapping to one particular instance of the basic unit of concurrency of the Ada language (i.e. the task). Conversely, protected objects resemble active objects in so far as having control over when their invoked operations are executed, but, unlike them and in accordance with the corresponding language abstraction, need no independent thread of control.

The HRT-HOOD Computational Model requires that the threads of control associated with active objects abide by the following rules:

- threads may access shared data in a protected manner by means of mutually exclusive calls to dedicated resource servers, which are modelled as HRT-HOOD protected object;
- resource servers may offer a variety of services, each denoted by one distinct execution request; the execution requests exported by a resource server must ensure mutual exclusion to
concurrent callers and must be internally non-blocking (i.e. allow no functional activation constraints nor perform blocking calls, for either would render static analysis impossible);

- threads may synchronise with one another by means of mutually exclusive calls to dedicated synchronisation servers;

- synchronisation servers allow threads to suspend on synchronisation calls and other threads to perform non-blocking triggering calls; the model requires that any thread which wishes to suspend on a synchronisation call be allowed to do so exclusively through the services of a dedicated synchronisation server. (In other words, no two threads can suspend on one and the same synchronisation call so that no suspension queue be required for any synchronisation service.)

HOOD objects are defined by the service they provide for use by other objects. The provided service is specified in the interface of the object. The control behaviour of the object with respect to the execution requests received at the interface is described in the object control structure (OBCS). The relevant services are implemented by a set of operation control procedures (OPCS). HRT-HOOD retains these base principles and maps the resulting entities on to the abstractions supported by the Computational Model we have just described.

Cyclic objects do not provide external operations that can be invoked by other objects, except for those which request the termination of the current operation of the object or the reset (e.g.: mode change) of its service status. Accordingly, the cyclic object is implemented by a maximum of two cooperating entities, as follows:

- the thread, which is to perform the time-triggered operation of the object and maps to an independent thread of control implemented as an entry-less Ada task with a single, time-triggered invocation event and a potentially unbounded number of invocations; and

- the OBCS, which is needed only if the object is required to support execution requests associated with termination or reset operations. The communication between the cyclic thread and its OBCS can be asynchronous (and, therefore, implemented as an Ada 95 asynchronous transfer of control operation initiated by the OBCS and taking effect in the thread) or synchronous (and, consequently, implemented as a synchronous status enquiry performed by the thread towards the OBCS). In the latter case (which is, in fact, the preferred one) the OBCS of the cyclic object naturally maps to a resource server.

Sporadic objects do always support at least one execution request, which triggers the sporadic operation of the object. The triggering event may be external (i.e. an interrupt) or internal (i.e. on demand from another software object in the system). The object triggered by the former type of execution request is called interrupt sporadic; the object triggered by the latter, software sporadic. Similarly to cyclic objects, sporadic objects may also provide additional execution requests for termination or reset operations.

The software sporadic object is modelled by a maximum of two cooperating entities, as follows:
- the **thread** in charge of the sporadic operation of the object, which maps to an independent thread of control implemented as an entry-less Ada task with a single, event-triggered invocation event and a potentially unbounded number of invocations; and

- the **OBCS**, which is to receive the execution requests arriving from other objects in the system and dispatch them to the sporadic thread. The triggering event is transferred from the OBCS to its software sporadic thread as the release from a dedicated synchronisation call. All other communications follow the model described for the cyclic object. As a result of that, the OBCS of the software sporadic object naturally maps to a **synchronisation server**.

Figure 4.5 portrays a simplified view of the task templates required to model the cyclic thread, the software sporadic thread and the relevant OBCS (synchronisation server).

<table>
<thead>
<tr>
<th>task Cyclic is</th>
<th>task Software_Sporadic is</th>
</tr>
</thead>
<tbody>
<tr>
<td>pragma priority (value);</td>
<td>pragma priority (value);</td>
</tr>
<tr>
<td>end Cyclic;</td>
<td>end Software_Sporadic;</td>
</tr>
<tr>
<td>task body Cyclic is</td>
<td>task body Software_Sporadic is</td>
</tr>
<tr>
<td>-- local variables</td>
<td>-- local variables</td>
</tr>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>-- set time reference T</td>
<td>-- main loop external set up</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>delay until T;</td>
<td>OBCS.Wait (params);</td>
</tr>
<tr>
<td>-- periodic action</td>
<td>-- sporadic action</td>
</tr>
<tr>
<td>T := T + Period;</td>
<td>end loop</td>
</tr>
<tr>
<td>end loop;</td>
<td>end Software_Sporadic;</td>
</tr>
<tr>
<td>end Cyclic;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>protected OBCS is</th>
<th>protected body OBCS is</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry Wait (params);</td>
<td>-- local variables</td>
</tr>
<tr>
<td>procedure Signal (params);</td>
<td>entry Wait (params)</td>
</tr>
<tr>
<td>end OBCS;</td>
<td>when (Barrier) is</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>-- release actions</td>
</tr>
<tr>
<td></td>
<td>-- raise barrier</td>
</tr>
<tr>
<td></td>
<td>end Wait:</td>
</tr>
<tr>
<td></td>
<td>procedure Signal (params) is</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>-- lower barrier</td>
</tr>
<tr>
<td></td>
<td>end Signal:</td>
</tr>
<tr>
<td></td>
<td>end OBCS;</td>
</tr>
</tbody>
</table>

Figure 4.5: Task Templates.

This modelling paradigm does not necessarily hold for the implementation of the **interrupt sporadic** object, as the service it provides is typically immediate and short-lived and, hence, has no use for termination operations while having an implicit (hardware) caller for the triggering operation. Depending on the support provided by the underlying implementation language, the interrupt sporadic object may map to an interrupt task (like in Ada 83) or a protected interrupt procedure (like in Ada 95).

Whereas interrupt sporadic objects are a powerful aid for the designer to explicitly model the service associated with the arrival of external interrupts (which are an essential ingredient of any real-time embedded system) software sporadic objects represent a very flexible tool to express and model **precedence activation constraints** between concurrent activities. As discussed in chapter 2, in fact, precedence activation constraints frequently arise between several of the data processing functions which are performed on board. In particular, intelligent use of software sporadic objects, in conjunction with suitable assignment of software priorities, caters for the achievement of the **increased responsiveness** and **maximum utilisation of available processing power** requirements discussed in chapter 1 (R.4 and R.6 respectively).
4.5.2 Ada Implementation

The engineering concept depicted in figure 4.2 was expressly intended to operate with Ada 95 software production factories. However, while awaiting industrial-quality commercial implementations of the language standard, the current implementation of the relevant industrial toolset has been realised upon an existing Ada 83 technology baseline augmented by the introduction of a limited number of upward-compatible enhancements. Implementations of the proposed concept based on actual Ada 95 technology will be realised as soon as mature technology becomes available. (Attractive concept demonstrators may possibly be implemented by integration of such handy technology as GNAT [Ada Core Technologies, 1996] and STAMP [Chapman et al., 1996].)

Implementation of the HRT-HOOD Computational Model on Ada 83 technology requires the following distinct forward-compatible enhancements:

- a large range of software priorities (≥ 64);
- the concept of passive task with similar semantics to the protected type of the Ada 95 language standard [ISO, 1995];
- the concept of absolute delay to permit jitterless modeling of periodic tasks;
- the concept of monotonic time to avoid the overhead of the time-of-day based clock handling.

Our Ada implementation was targeted to the 32-bit Embedded Real-Time Computing Core (ERC32) [Gaisler, 1994, ESA, 1992a, Saab Ericsson Space, 1997], a SPARC v7 based chipset inclusive of Integer Unit, Floating Point Unit and Memory Controller, intended for use in nocache no-virtual memory single-board computers for advanced new-generation on-board systems. Amongst other features, which are not discussed in this thesis, the ERC32 chipset and associated Ada compilation system optionally supports board configurations which include the ATAC (Ada Tasking Coprocessor) chip [Roos and Gomez-Moliner, 1992]. The ATAC is a memory-mapped hardware device which performs Ada 83 tasking operations on behalf of classical software runtime systems. Reasons of source-level compatibility between systems built for configurations with or without the ATAC dictated the use of Ada 83 interrupt tasks to model interrupt sporadic objects.

We have recalled earlier in section 4.2.3 that our notion of Computational Model entails the definition and the timing characterisation of the concurrency management mechanisms used at run time for the execution of the system.

In the following, we provide a brief description of the tasking primitives which contribute to the determination of the run-time scheduling behaviour of systems built in accordance with the HRT-HOOD Computational Model. Subsequently, we provide a timing bound for the execution of those primitives on our reference target platform. This shows that the run-time system required to support the HRT-HOOD Computational Model is actually small, compact and fully characterised.

Cyclic tasks call primitive Delay Until to command the time of their next release and the wake-up system uses an Interval Timer instead of the conventional periodic clock. The overall
worst-case execution time of the primitive results from the sum of two values: the placement of
the task control structure in the interval time queue (Delay Until(Enter)) and the return from the
call upon release (Delay Until(Exit)).

Interrupts off the Interval Timer are serviced by primitive Timer_Int. Primitive Ready changes
the released cyclic tasks’ status to ready. On modifications to the ready status list, primitive Select
is invoked to determine the "best-task-out"; this may incur preemptive switch to a new running
task, which is performed by primitive Switch.

Primitive Int_Handling initiates an interrupt accept statement in the body of the designated
interrupt sporadic task, while Int.Wait(Enter) and Int.Wait(Exit) allow control to respectively
enter and leave the interrupt accept body.

PO_Entry and PO_Exit control respectively the access to and the release of server tasks and
include the relevant raising and lowering of the caller’s priority.

The implementation of the blocking call to synchronisation servers uses a primitive semaphore
structure: the software sporadic task’s call to the server’s guarded entry translates into the caller’s
suspension on the primitive semaphore (Sem.Wait(Enter)). Arrival of the releasing call causes the
suspended task to be freed from the semaphore’s queue (Entry.Queue_Mgmt) (which includes the
call to Sem.Signal) exit from the suspensive call (Sem.Wait(Exit)) and potentially become the new
running task.

Primitive Select involves queue management operations which are typically exposed to pes-
simistic bounds; the problem was circumvented by redesigning the primitive so as to preserve
minimal execution time and also achieve low worst-case bounds. The design restriction of having
at most one software sporadic task wait on any given synchronisation server’s semaphore queue
allows all of the relevant primitive operations to be easily bounded. All the other primitives in
the list exhibit deterministic execution time bounds. The characterisation of all such bounds on
execution on the selected platform is stored in the so-called run-time system characterisation file.

Table 4.4 displays the results of a preliminary characterisation of the execution cost of the
tasking primitives in our Ada run-time system. The measurements were performed on an ERC32
demonstration board upon completion of the implementation of the relevant Ada technology. The
measurements were taken on board configurations with as well as without the ATAC and included
the determination of the longest deferred-preemption time incurred during run-time system op-
eration; as it will be explained in section 4.5.3, in fact, that value contributes to the determina-
tion of blocking overhead. The table also indicates the correspondence between the tasks in our
Computational Model and the run-time primitives required to support them. (Note that the term
"sporadic" in the table denotes the software sporadic task, whereas the interrupt sporadic task is
denoted by the term "interrupt".)

All expressions in table 4.4 are constant except for the one which describes the cost of placing
a cyclic thread in the delay queue at a position lower than the top in the non-ATAC version of the
system. The actual position depends on the relative ordering of the required awake time by the
Interval Timer. Term C, thus, denotes the total number of cyclic threads currently placed ahead
of the presently suspending thread. The best value for C, for use by static analysis, obviously
depends on the knowledge available to the tool in question. In the case of our analysis model, this
value is defaulted to the total number of cyclic threads in the system. Conversely, in the ATAC

74 Development of On-Board Embedded Real-Time Systems: An Engineering Approach
version of the system, this value remains constant as the thread queuing is taken care of entirely by the ATAC chip itself.

<table>
<thead>
<tr>
<th>RTS Primitive</th>
<th>Used by / for</th>
<th>DEM32 (10 MHz 0 Wait-states)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>non-ATAc run-time</td>
<td>ATAC run-time</td>
</tr>
<tr>
<td>PO_Entry</td>
<td>cyclic, sporadic, interrupt</td>
<td>8.0</td>
<td>13.8</td>
</tr>
<tr>
<td>PO_Exit</td>
<td>cyclic, sporadic, interrupt</td>
<td>11.0</td>
<td>11.0</td>
</tr>
<tr>
<td>Int.Wait (Enter)</td>
<td>interrupt</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Int.Wait (Exit)</td>
<td>interrupt</td>
<td>3.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Sem.Wait (Enter)</td>
<td>sporadic</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>Sem.Wait (Exit)</td>
<td>sporadic</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Entry_Queue_Mgmt</td>
<td>sporadic</td>
<td>6.0</td>
<td>8.0</td>
</tr>
<tr>
<td>Select</td>
<td>cyclic, sporadic, interrupt</td>
<td>5.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Switch</td>
<td>cyclic, sporadic, interrupt</td>
<td>34.0</td>
<td>41.0</td>
</tr>
<tr>
<td>Delay Until (Insert at Top)</td>
<td>cyclic</td>
<td>39.0</td>
<td>23.0</td>
</tr>
<tr>
<td>Delay Until (Insert Lower)</td>
<td>cyclic</td>
<td>22.0 + C * 3.0</td>
<td>23.0</td>
</tr>
<tr>
<td>Delay Until (Exit)</td>
<td>cyclic</td>
<td>8.0</td>
<td>8.0</td>
</tr>
<tr>
<td>Timer_Int</td>
<td>cyclic</td>
<td>21.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Ready</td>
<td>cyclic</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Int.Handling</td>
<td>interrupt</td>
<td>67.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Max.Deferred_Preemption</td>
<td></td>
<td>130.0</td>
<td>65.0</td>
</tr>
</tbody>
</table>

### 4.5.3 Static Analysis

The key asset of the engineering approach shown in figure 4.2 lies in the ability to support the design and implementation of a system which can be statically analysed for its real-time characteristics at all stages of development. We have claimed earlier in this chapter that this ability descends from the adoption of the notion of Computational Model as the "driving force" of the design process. We now want to present the constructive elements of our approach to static analysis and the way these provisions feed the iterative and incremental component of our development strategy.

#### Foundations of Response Time Analysis

The static analysis model chosen for our engineering concept aims at the prediction of worst-case response times (cf. e.g.: [Joseph and Pandia, 1986, Audsley et al., 1993]). The model stipulates that one thread’s worst-case response time be defined as the longest elapsed time it takes for that thread to complete its most demanding set of activities in response to an activation occurring under maximum contention from the rest of the system. (The term thread is used in the following as a synonym for task.) The worst-case response time of any thread \( \tau_i \) does, thus, result from suitable combination of the following three distinct components:

(i) The worst-case computation time of thread \( \tau_i \), \( WCCT_i \), which is defined as the sum of the time cost of all \( \tau_i \)’s sequential blocks of execution which lay in the statically determined worst-case path enclosed within the thread’s main loop (the thread’s execution profile) in addition to the
time cost of the run-time system services required for the support of that execution.

(ii) The interference incurred by \( \tau_i, I_i \), which is caused by the occurrence of preemptive execution of higher-priority threads and higher-priority run-time system services incurred during \( \tau_i \)'s ready period; in the HRT-HOOD Computational Model, the interference incurred from run-time system operation is limited to the handling of the interrupts off the Interval Timer. All other interrupts are, in fact, tied to the activation and execution of interrupt sporadic tasks and, therefore, contribute to the response time of the relevant task.

(iii) The blocking experienced by \( \tau_i, B_i \), which originates from the possibility that a due release of \( \tau_i \) be delayed by other effects than those arising from preemptive interference; such effects occur when the run-time system protects the execution of internal critical sections by temporarily inhibiting (i.e. deferring) preemption as well as a consequence of using priority ceiling emulation for the implementation of mutual exclusion in the communications between tasks and servers; use of priority ceiling emulation may, in fact, delay the release of tasks whose priority is higher than the caller but lower than the server's ceiling; response time analysis prescribes that worst-case blocking be determined as the largest possible delay effect incurred from any of the two sources.

For any thread \( \tau_i \), component WCCT\(_i\) is fully determined at compile time on the Ada closure of the program, component \( B_i \) is a function of the assigned priorities and the system's run-time performance, and component \( I_i \) is a function of the system load.

Component WCCT\(_i\) and \( B_i \) are maximised by analysis. Care must be taken, though, to avoid incurring excessive pessimism in their determination, as this may hinder the usefulness of the analysis. The approach taken to the determination of \( B_i \) and the issues in the generation of the worst-case execution profiles from which WCCT\(_i\) is determined are discussed separately in the following.

Component \( I_i \) is maximised by assuming all runs to occur under the notional concept of critical instant, whereby:

- all cyclic tasks are assumed to be disjointly released at time \( t_0 = 0 \)
- all interrupt sporadic tasks are assumed to be disjointly triggered at time \( t_0 = 0 \) and arrive at their maximum frequency
- all software sporadic tasks are assumed to be disjointly released off their synchronisation server's queue at time \( t_0 = 0 \).

The formulae which capture the interference effects on \( \tau_i \)'s ready period over the interval \([0, t]\) are shown in the following, where notation \( j \in HP(i) \) denotes that thread \( \tau_j \)'s priority is greater than \( \tau_i \)'s (i.e. \( Pr(\tau_j) \geq Pr(\tau_i) \)) notation \( j \in LP(i) \) denotes the converse (i.e. \( Pr(\tau_j) < Pr(\tau_i) \)) and \( T_j \) denotes \( \tau_j \)'s period (for cyclic threads) or minimum interarrival time (for sporadic threads):

**Interference from Preemptive Execution of Higher-Priority Threads over \([0, t]\):**

\[
I_i = \sum_{j \in HP(i)} \left\lfloor \frac{t}{T_j} \right\rfloor WCCT_j
\]  

(4.1)
Interference from Interrupts Off the Interval Timer over \([0,t)\):

\[
C_i^t = K_i^t \cdot (\text{Timer}_\text{Int} + \text{Ready} + \text{Select})
\]  \hspace{1cm} (4.2)

The terms in *italics* in equation 4.2 denote the run-time primitives involved in the readying of a (cyclic) thread off the delay queue. The constant values to use for the resolution of these equation are those listed in table 4.4. Term \(K_i^t\) determines how many times the Interval Timer triggers over the time span under consideration. This number is determined as follows:

in ATAC mode (cf. equation 4.3) the ATAC filters out all Timer interrupts and *only* releases those which induce preemption:

\[
K_i^t = \sum_{j \in HP_{\text{CYCLIC}}(i)} \left\lfloor \frac{t}{T_j} \right\rfloor
\]  \hspace{1cm} (4.3)

conversely, in non-ATAC mode (cf. equation 4.4) there occurs no filtering and the calculation needs to consider also the *first* release of all the lower-priority cyclic threads assumed to disjointly occur around the critical instant:

\[
K_i^t = \sum_{j \in HP_{\text{CYCLIC}}(i)} \left\lfloor \frac{t}{T_j} \right\rfloor + \sum_{k \in LP_{\text{CYCLIC}}(i)} 1
\]  \hspace{1cm} (4.4)

No further releases of lower-priority threads may occur before \(\tau_i\) completes its critical-instant activation.

The analysis devices foreseen in our engineering concept support two variants of analysis techniques. The first variant, based on deadline monotonic (DM) theory [Audsley et al., 1991, Audsley et al., 1993], assumes that tasks’ deadlines cannot exceed the respective period (or minimum interarrival time) and determines, for every individual thread, the response time for a *single* critical-instant release. The other variant, based on the extended version of deadline monotonic theory presented in [Tindell et al., 1994], referred to as arbitrary deadline (AD) assumes that deadlines may be arbitrarily greater than the relevant period (or minimum interarrival time) and, therefore, extends the solution space to *multiple, overlapping* releases of a task. The critical-instant assumptions are, thus, worsened by tasks’ releases being delayed past their due time also by the outstanding completion of their previous releases.

The equations for DM and AD response time analysis are shown in the following. Both are based on recurrence relations in which thread \(\tau_i\)'s response time, \(R_i^n\), is expressed as a monotonically increasing summation term. The DM recurrence is somewhat simpler than its AD variant and guaranteed to converge when the system’s utilisation is not greater than 1. The AD variant, albeit based on the same conceptual model as DM, is slightly more complex as its solution space extends across *multiple, overlapping* releases; as shown by equation 4.12, the search stops as soon as the response time for the last release of the thread no longer overlaps the next due activation.
Response Time (DM):

\[ R_i^n = B_i + WCCT_i + I_i^{R_i^{n-1}} + C_i^{R_i^{n-1}} \quad (n > 1) \quad (4.5) \]

\[ R_i^1 = B_i + WCCT_i \quad (4.6) \]

Response Time (AD):

*Busy Window at \((q+1)th\) release:*

\[ R_i^{n+1}(q) = B_i + (q + 1)WCCT_i + I_i^{R_i^q}(q) + C_i^{R_i^q}(q) \quad (q, n \geq 0) \quad (4.7) \]

where:

\[ R_i^0(q) = R_i(q - 1) \quad (q \geq 1) \quad (4.8) \]

and

\[ R_i^0(0) = B_i + WCCT_i \quad (4.9) \]

*Response Time at \((q+1)th\) release:*

\[ R_i(q) = R_i^q(q) - qT_i \quad (4.10) \]

*Worst-Case Response Time:*

\[ R_i = \max_{q \in N} R_i(q) \quad (4.11) \]

where:

\[ N = \{q\} : R_i^q(q) > (q + 1)T_i \implies R_i(q) > T_i \quad (4.12) \]

**Blocking Overhead Determination**

The worst-case blocking effect incurred on one thread’s release is determined as the largest value between the single longest period of run-time deferred preemption and the longest-duration entry call to a higher-precision server performed by a lower-priority task.

The former value is a constant characteristic of the run-time system implementation. In the case of our Ada implementation, this value is minimised by the restrictive nature of the HRT-HOOD Computational Model.

The latter value is a variable thread-specific attribute which depends upon characteristics of the overall application, such as the assigned priorities and the performance of servers’ entries. The pessimism potentially embodied in the determination of this value is minimised by the analysis device discussed in the following.

In the HRT-HOOD Computational Model, calls to server entries conform to any of the types shown in Table 4.5:

<table>
<thead>
<tr>
<th>server type</th>
<th>call type</th>
<th>call denotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource server (RPO)</td>
<td>unguarded service call</td>
<td>RPO.Service</td>
</tr>
<tr>
<td>synchronisation server (SPO)</td>
<td>unguarded releasing call</td>
<td>SPO.Signal</td>
</tr>
<tr>
<td>synchronisation server (SPO)</td>
<td>guarded suspensive call</td>
<td>SPO.Wait</td>
</tr>
</tbody>
</table>

Table 4.5: Types of PO Calls.
The contribution to one thread’s blocking overhead resulting from use of priority ceiling emulation is, thus, determined by the single largest execution cost of any of the calls listed in table 4.5 in accordance with the definition given in section 4.5.3. This overhead includes the WCCT of the relevant protected service and the execution cost of all the run-time system operations possibly involved in the execution of that service.

Let us now look at the individual server calls in detail. Calls to resource servers are unconditional, hence incur a constant run-time system overhead. Calls to synchronisation servers are guarded, hence exhibit worst-case and best-case execution profiles: a guarded suspensive call may find the guard open (best case) and incur no suspension, or closed (worst case) and incur suspension and deschedule; an unguarded releasing call (SPO.Signal) may find no awaiting task in the entry queue (best case) and let the caller continue undisturbed, or one awaiting task (worst case) and cause its release off the queue, the execution of the wait service and, eventually, the potential deschedule of the signaller depending on the relative base priority of the waiter.

Table 4.6 lists the individual overhead components which are incurred on the execution of the PO calls shown in table 4.5 under both worst and best case. The terms in italics in the table denote the constant values associated with the relevant primitives as listed in table 4.4.

Notably, table 4.6 shows that the worst-case execution of an SPO.Wait call is made up of two components that occur at two separate points in time, as follows:

- the prologue, identified as component (3.1) in the table, which occurs from the waiter’s request to enter the SPO until waiter’s suspension and placement in the entry queue; and

- the epilogue, identified as component (3.2) in the table, which occurs from waiter’s release from the entry queue until waiter’s departure from the SPO and always directly follows the execution of component (2) by the relevant signaller.

<table>
<thead>
<tr>
<th>call type</th>
<th>execution components</th>
<th>id</th>
<th>case type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPO.Service</td>
<td>PO_Entry + WCCT(Service) + PO.Exit</td>
<td>(1)</td>
<td>worst,best</td>
</tr>
<tr>
<td>SPO.Signal</td>
<td>PO_Entry + WCCT(Signal) +</td>
<td>(2)</td>
<td>worst</td>
</tr>
<tr>
<td></td>
<td>+ Entry.Queue.Mgmt + Switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPO.Wait (in)</td>
<td>PO_Entry + WCCT(guard.eval) +</td>
<td>(3.1)</td>
<td>worst</td>
</tr>
<tr>
<td></td>
<td>+ Sem.Wait(Enter) + Select + Switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPO.Wait (out)</td>
<td>Sem.Wait(Exit) + WCCT(guard.eval) +</td>
<td>(3.2)</td>
<td>worst</td>
</tr>
<tr>
<td></td>
<td>+ WCCT(Wait) + PO.Exit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPO.Signal (no.wait)</td>
<td>PO_Entry + WCCT(Signal) + PO.Exit</td>
<td>(4)</td>
<td>best</td>
</tr>
<tr>
<td>SPO.Wait (open)</td>
<td>PO_Entry + WCCT(guard.eval) +</td>
<td>(5)</td>
<td>best</td>
</tr>
<tr>
<td></td>
<td>+ WCCT(Wait) + PO.Exit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The execution pattern of the prologue and the epilogue component of the SPO.Wait call directly emanates from prescriptions of the HRT-HOOD Computational Model and may effect the worst-case blocking experienced by signallers upon call of (2). Our model does not allow tasks to share the same priority level. In keeping with this prescription, the ceiling priority of server tasks is set to (at least) one level higher than the maximum priority of their callers. Hence, in the worst-case SPO scenario, as soon as the signaller relinquishes the SPO on completion of (2), it is
preempted by the waiter which, once kicked off the entry queue, is assigned the ceiling priority of the SPO and can, therefore, execute (3.2). (This explains the Switch component in the breakdown of (2).) This phenomenon must be taken into account in the determination of the priority ceiling blocking experienced by the signaller tasks in the system, as follows:

- if the base priority of the waiter (w) is higher than the signaller’s (s) i.e. weHP(s), the execution cost of (3.2) is captured by the interference incurred by the signaller on its worst-case execution (I_s);

- otherwise, s experiences an extra blocking overhead, equal to the execution cost of (3.2) plus the Select and Switch service components required to restore its execution context; that extra blocking overhead is incurred for every (2) call performed in the worst-case execution profile for which weLP(s); the resulting overhead adds to the general priority ceiling blocking for that thread, which is determined in the manner described in the following.

Response time analysis based on DM considers one single critical-instant release of the thread subject of analysis under the run-time conditions which maximise the possible source of contention and interference. With reference to table 4.6, therefore, DM is interested in the worst-case service values only. Conversely, analysis based on AD contemplates multiple, potentially overlapping releases and may, therefore, need to consider best-case values, too. In fact, for example, a software sporadic thread τ_i performing a guarded suspensive call at its q-th release (with q > 0) will find the guard open if τ_i’s priority is lower than the releasing thread’s. Similarly, thread τ_i performing an unguarded releasing call will find no awaiting thread in the entry queue if τ_i’s priority is higher than that of the software sporadic thread associated with that synchronisation server.

Table 4.7 prescribes how the individual PO call overhead components listed in table 4.6 contribute to the determination of the bound for the priority ceiling blocking experienced by thread τ_i.

<table>
<thead>
<tr>
<th>Pr(Caller) vs Pr(τ_i)</th>
<th>RPO.Service</th>
<th>SPO.Signal</th>
<th>SPO.Wait</th>
<th>blocking factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>(1)</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>L</td>
<td>H</td>
<td>max(1,2)</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>max(1,2,3,2,3.1)</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>max(1,2,3,2,3.1)</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>(3.1)+3.2</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>(2)</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>max(1,(3.1)+3.2)</td>
</tr>
</tbody>
</table>

The information stored in the table is to be interpreted as follows:

let us assume that J denotes the thread set captured by equation 4.13:

\[ \{\tau_j\}_{j \in J} : (j \neq i) \land (j \in LP(i)) \land (j \Rightarrow P.O.E) \land (P.O \in HP(i)) \]  (4.13)
where the notation \( j \implies PO.E \) denotes that \( \tau_j \)'s WCET profile includes a PO call (PO.E) of the type indicated by the relevant column header in the table;

entries tagged L in columns 2-4 denote then the fact that:

\[
\exists k : k \in J \land (WCCT_k(E) = \max_{j \in J}(WCCT_j(E)))
\]

whereby the PO.E call made by \( \tau_k \) represents the largest priority ceiling blocking caused on \( \tau_i \) by PO calls of that type; conversely,

entries tagged H indicate that the thread set \( J \) is empty for that particular PO call type and, consequently, \( \tau_i \) experiences no priority ceiling blocking from PO calls of that type.

**Characterisation of Task Management Overhead**

The information contained in table 4.4, in conjunction with the analysis of the Ada implementation discussed earlier in this section, allows the run-time system contribution to one thread's WCCT to be fully characterised. In fact, table 4.8 uses the constant values listed in table 4.4 to describe (and bound) all of the task management and administration services incurred by individual tasks under the initial conditions required for DM and AD analysis.

<table>
<thead>
<tr>
<th>task / event</th>
<th>analysis case</th>
<th>DM &amp; AD ((g = 0))</th>
<th>AD ((g &gt; 0))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cyclic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>on release</td>
<td>Switch + Delay Until(Exit)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>on suspension</td>
<td>Delay Until(Enter) + Select + Switch</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>Int Sporadic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>on release</td>
<td>Int._Handling + Select + Switch + Int._Wait(Exit)</td>
<td>idem</td>
<td></td>
</tr>
<tr>
<td>on suspension</td>
<td>Int._Wait(Enter) + Select + Switch</td>
<td>idem</td>
<td></td>
</tr>
<tr>
<td><strong>Sw Sporadic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>on release</td>
<td>SPO.Wait (out) if ( \text{we} HP(s) )</td>
<td>idem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 if ( \text{we} LP(s) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>on suspension</td>
<td>SPO.Wait (in)</td>
<td>SPO.Wait (in) if ( \text{we} HP(s) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPO.Wait (open) if ( \text{we} HP(s) )</td>
<td>SPO.Wait (open)</td>
<td></td>
</tr>
<tr>
<td><strong>Any Task Type</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPO.Service</td>
<td>as per table 4.6</td>
<td>idem</td>
<td></td>
</tr>
<tr>
<td>SPO.Signal</td>
<td>SPO.Signal(no.wait) if ( \text{we} HP(s) )</td>
<td>SPO.Signal(no.wait)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPO.Signal + SPO.Wait(out) + Select + Switch if ( \text{we} LP(s) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Interval Timer</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>on cyclic release</td>
<td>( K^* ) (Timer.Int + Ready + Select)</td>
<td>idem</td>
<td></td>
</tr>
</tbody>
</table>

General task execution overhead obviously includes those resulting from the server call services listed in table 4.6 for every such call retained in the worst-case profile of the task. In particular, in force of the discussion in section 4.5.3, the critical-instant assumptions established...
in section 4.5.3 for the determination of the task execution overhead shall be refined by requiring that every waiter (software sporadic) task $w$ be considered as:

- released off its entry queue at time $t_0 = 0$ in the respect of every thread $\tau_i : w \in HP(\tau_i)$; and

- kicked off its entry queue by its respective signaller, for every signaller task $s : w \in LP(s)$.

**Worst-Case Execution Time Profile Generation**

One distinguishing feature of our baseline technology resides with the implementation of a WCET profile generator capability *tightly integrated* with the Ada compilation system. The required operation of this capability is described in [Thomson Software Products, 1995].

The function of the WCET profile generator is to determine, for every thread in the system, the execution profile which controlledly maximises the WCCT component for that thread. As indicated earlier in section 4.5.3, thread $\tau_i$'s WCCT$_i$ is made up of two components: (i) the sum of the time cost of all the sequential blocks of execution which lay in the path statically determined by the WCET profile generator within $\tau_i$'s main loop; and (ii) the run-time system overhead incurred by $\tau_i$ in the execution of the selected path. The latter component is statically determined on the thread's type and the server calls retained in the profile. The relevant overhead components are listed in table 4.8. Hence, the function of the WCET profile generator is to provide a bound for the former part of the thread's WCCT.

The design of our WCET profile generator is relatively simple and takes advantage of the fact that the designated target hardware is a single-board computer based on an ERC32 chipset [ESA, 1992a, Gaisler, 1994, Saab Ericsson Space, 1997] provided with *no cache* and *no memory hierarchy*. The present version of the WCET tool as described in the following was expressly designed to serve the role of "concept demonstrator", which it has in fact successfully accomplished. Our future plan is to push further the performance of the tool (particularly as regards the control of pessimism) and progressively refine its implementation and operation using the guidance of other important research work in the domain, e.g.: [Park, 1993, Puschner and Koza, 1989, Chapman et al., 1996].

The WCET profile generator presently operates in two phases:

- The *compile-time* phase stores in the program library, for every compiled subprogram, the call graph of *basic blocks* created by the code generator. (The basic block being the largest set of sequential instructions with no internal flow of control.) The basic block timing information is used to decorate the abstract syntax tree which, in contrast with conventional technology, is not disposed on exit from the front-end but retained to guide the WCET processing of the subsequent phase.

- The *bind-time* phase, which operates on the *closure* of the supplied Ada program, recognises the set of *legal* threads which constitute the program (flagging the presence of any illegal ones) and performs, for each of them, the traversal of the relevant set of call graphs for the determination of the worst-case path. Legal threads are those which comply with the required task template and do not perform difficult-to-bound operations, such as:
(1) direct and indirect recursion
(2) direct and indirect heap management
(3) dynamic task creation and deletion

The generation of the worst-case execution profile from the application source code must attempt to capture both the local worst-case at thread-level and the global worst-case at application-level, in a manner which incurs a controlled degree of induced pessimism.

Excessive pessimism may arise, for example, when the resolution of a branch or the bounding of an iteration within one thread's profile fail to capture application-wide path exclusion conditions (e.g.: mutually exclusive operating modes) or run-time best-bounding information. This may cause otherwise provably impossible paths to be selected and consequently yield too conservative predictions.

The execution profile generator mitigates the effect of such problems by providing means for the user to annotate the source code with a loop-bound and a path-exclusion pragma, as follows: (i) \texttt{pragma Loop.Count (< constant >)} placed before a for or while loop construct allows the user to supply the preferred bound to an otherwise unbound iteration; the compiler uses the provided bound value to cost the iteration but returns warnings if it was able to statically determine a better bound; (ii) \texttt{pragma Exclude.Wcet} placed inside a conditional branch, procedure body or task body causes the exclusion of the tagged construct from the selected path.

For the sake of the practicality of use of the present release of the tool, exception handlers and basic blocks containing an explicit raise instruction are implicitly excluded from the WCET path. The rationale behind this choice is that our baseline Ada technology did not ensure bounded overhead for the determination of the handler to be associated with the raised exception. We regard the assurance of a bounded (and acceptably low) overhead for this operation as an essential pre-requisite for the handling of exceptions (whether pre-defined or user-defined) to be considered in the generation of the WCET execution profile. Future releases of this tool will include this ability.

In actual fact, the achievement of justified maximisation of WCCT_i is not the sole objective of the WCET profile generator. There, in fact, exist two distinct ways for thread \( \tau_i \) to effect system's responsiveness:

- a longer WCCT_i induces a longer I_j on thread \( \tau_j \forall j : j \in LP(i) \);
- a PO.Call performed by \( \tau_i \) may contribute to B_k for thread \( \tau_k \forall k : k \in HP(i) \land k \in LP(PO) \).

The WCET profile generation algorithm must, therefore, also seek to achieve justified maximisation of B_k.

Consider the code fragment in example 4.5.1 and assume that \texttt{< condition >} cannot be statically resolved. The example shows a classical case in which local maximisation of WCCT_i may degrade the determination of B_k for any thread \( \tau_k \) in the system. This case occurs on the selection of branch B, when no trace of PO.Call is retained in the thread's profile.

The problem is resolved, in our model, by instructing the branch selection algorithm to keep record of all the server calls performed outside the retained profile and to require that alternate
if <condition> then
  -- branch A
  <A1> -- sequential block
  P0.Call
  <A2> -- sequential block
else
  -- branch B
  <B> -- sequential block
end if;

Example 4.5.1: Effect of Blocking on Path Selection.

blocking analysis be performed of the potential blocking effect of such calls. This analysis may possibly yield a larger $B_j$ value for some thread $\tau_j$, thereby highlighting a potential conflict between local and global worst-case path selection criteria. In any such case, the server call responsible for thread $\tau_j$'s alternate blocking is identified to the user, who is advised to consider repeating the analysis forcing the extraction of WCET execution profiles which include that server call.

Feedback to Design

As long as the user maintains the design information current with the progress of the implementation, the semantic contents of an HRT-HOOD description allows the designer to perform static timing analysis of the system and to derive the feedback necessary to steer design and implementation to completion.

It is central to our engineering concept that the designer be able to start performing this type of analysis very early in the development life cycle. To this end, we enable the designer to actively effect the determination of the WCET profile of the system. The designer may do so by supplying override values for named subprograms via special directives submitted to the WCET profile generator along with the program source. This provision allows the designer to start performing informative analyses of the timing behaviour of the system since as early as the establishment of the initial design skeleton.

The earliest skeleton of an HRT-HOOD system is typically comprised of as little as the "use" relationships placed between objects and the abstract outline of the objects' operation control structure (OPCS). In our model, the designer can decorate that initial skeleton with the expected timing of the object operations, without having to necessarily provide all of the relevant code. The measurements overridden by the designer-supplied values may thus well be nil, when the actual subprogram code is still to be supplied, or else a partially representative value, as the subprogram code is still incomplete. Along with the consolidation of the design and the provision of the real code of the application, the designer would then progressively remove the override values and replace them by the timing estimates actually computed by the WCET profile generator.

This feature facilitates the occurrence of design iterations triggered by the interpretation of feedback from static timing analysis (which is intrinsically relevant to the physical view of the
system) to initiate from as early in the development as the initial design skeleton (which typically corresponds to the baseline definition of the logical model of the system) and to occur within one and the same HOOD-based design framework, as shown in figures 4.1 and 4.2.

The key elements of information obtained from static analysis include: (1) the confirmation of the schedulability status of the individual threads of the system with respect to the relevant real-time requirements; and (2) the sensitivity of the system timeliness to variations in the processing requirements of individual threads. The results obtained for category (1) indicate, for every thread: the priority level automatically assigned to that thread; the response time determined for that thread and its WCCT and B break-down components; and the adjudged cause of the worst-case blocking incurred by that thread. The results obtained for category (2) indicate what variations in one thread’s WCCT would be permitted without affecting the schedulability status of the system.

4.6 Analysis of the Proposed Solution

This chapter has proposed an innovative evolutionary approach to the development of new generation software intensive on-board real-time embedded systems. The definition of the proposed approach has been arrived at from analysis of the emerging requirements discussed in chapter 1 and the decision to adapt the current process model to the changing development scenario in the fashion presented in chapter 3.

The proposed approach responds to the following demands:

- the need to perform informed selection and enforcement of a given Computational Model as integral part of the design process;

- the availability of a characterisable, efficient and statically analysable implementation of a Computational Model which meets the demands discussed in section 3.3.2 and the application requirements presented in chapter 2;

- the ability to support the iterative and incremental development process which we advocate as the primary means to meet the requirements discussed in chapter 1.

The main virtue of the proposed concept is that, in keeping with the choice for a smoothly evolutionary approach, it builds upon such existing wide-spread technology as HOOD [HTG, 1993] and Ada [ISO, 1987], only adding to it a limited number of distinct enhancements. The proposed enhancements preserve the integrity of the originating design method and extend the expressive and semantic power of the outgoing language standard [ISO, 1987] in a fashion forward-compatible with its recent revision [ISO, 1995].

This chapter has further outlined the structure and operation of the enabling technology associated with our development concept. One specific toolset instance incorporating all of the required enabling technology has recently been built [Logica, 1997] in the frame of a work programme funded by the European Space Agency and conducted under the technical coordination of the author of this thesis.
The block diagram of our toolset is depicted in figure 4.6. The toolset was designed as an instantiation of the reference concept depicted in figure 4.2. Special emphasis was placed on the provision of effective support to the four design steps B.1-B.4, discussed in section 4.3.1, which constitute the essence of our development strategy.

The toolset is presently comprised of the following components:

- an augmented version of the HOOD design method, called HRT-HOOD and defined in [Burns and Wellings, 1995b], directly supportive of a Computational Model based on the deadline monotonic fixed priority preemptive scheduling theory developed by the Real-Time Group at the University of York [Audsley et al., 1991] and [Audsley et al., 1993];

- a standard Ada 83 [ISO, 1987] cross compilation system augmented with Ada 95 real-time features [ISO, 1995] and provided with: (i) a precise characterisation of the timing behaviour of the run-time support to the Computational Model, and (ii) built-in capabilities for the extraction of WCET profiles of the application components, the operation of which is described in [Thomson Software Products, 1995, Thomson Software Products, 1996];

- a static analysis tool [Spacebel Informatique, 1996, Logica, 1996] designed in accordance with [Joseph and Pandia, 1986] and [Audsley et al., 1991] to perform response time analysis of systems generated with the aid of the toolset; our process model facilitates the iterative execution of this analysis stage on successive increments of the system; the analysis device determines: (a) the priority level of the individual components of the system; (b) their current schedulability status; and (c) reports on the sensitivity of the current design to variations in the execution time of its components.
Figure 4.7 depicts the intent and distinguishing features of the development approach discussed in this chapter in comparison to a schematic view of the current practice.

![Diagram of proposed approach compared to current practice]

**Figure 4.7: A Schematic View of the Proposed Approach.**

Any development method can be regarded as a structured aid to bridge the distance between one problem and the proposed solution. The span between the problem and the solution shown on the axis of the diagram in figure 4.7 broadly reflects the relative proportion of the main phases of development of on-board embedded real-time systems. With reference to the V model shown in figure 3.1, the three main phases of development are broadly referred to as: design, which is comprised of SR and AD and typically represents 25-30% of the development; implementation, which corresponds to DD and UT and typically absorbs ± 15% of the development; and testing, which includes IT and ST and takes up the most part of the development effort, typically around 55-60%.

The lower portion of the diagram shows the span of the process actually covered by conventional HOOD-based design. The span is rather modest and, as noted earlier, completely disconnected from the conclusive part of the implementation phase as well as from the various threads of testing (those assisted by aid tools and those not).

Conversely, the higher portion of the diagram shows that the use of the HRT-HOOD method aims at covering a much greater span of development by providing for (i) a unified design framework to facilitate and assist the occurrence of repeated feedback-based iterations across the design, implementation and testing activities and (ii) the explicit definition and enforcement of design constructs and constraints which emanate from the reference Computational Model and include sufficient semantics to allow for static analysis of the system since the early stages of development.

This approach may possibly increase the conceptual distance between the typical formulation
of the problem (UR) and the proposed design formalism based on the HRT-HOOD method. The key assumption of this thesis, though, is that the projected benefits (support to design iterations, support for enforcement and verification of the real-time properties) largely exceed the incurred disadvantages.

This notwithstanding, it may well be the case that the use of other specification formalisms in earlier phases of development – closer to the problem formulation and yet "translatable" to an initial HRT-HOOD design – may be a very desirable option to consider. This issue is presently being investigated at the European Space Agency but its discussion falls outside the scope of the work presented in this thesis.

Chapter 5 shall further illustrate the operation of the proposed development approach by means of a simple demonstrative example.
Chapter 5

Illustrative Example

5.1 Introduction

In this chapter, we use a simple illustrative example to demonstrate the operation of the engineering approach presented in chapter 4.

The presentation of the example will progress across each of the four main development steps B.1-B.4 presented in section 4.3.1 and will use the instantiation of the associated enabling technology shown in figure 4.6. All the elements of the example presented in the following were produced using the enhanced Ada 83 technology [Logica, 1997, Thomson Software Products, 1996] introduced in section 4.5.3 and developed as part of [Saab Ericsson Space, 1997]. Accordingly, the example uses language constructs which are semantically equivalent to but syntactically different from those of Ada 95. Similarly, the syntax used to express the input and output of the static analysis tools [Logica, 1997, Spacebel Informatique, 1996] referred to in the example is the one supported by the technology developed in [Saab Ericsson Space, 1997].

5.2 Construction and Analysis of an HRT-HOOD System

The most prominent feature of any HRT-HOOD design lays in that it can be statically analysed for its timing characteristics at virtually all of stages of development. This is possible because the method is structured so as to elicit the definition and facilitate the extraction and verification of the design properties which determine the real-time behaviour of the system in accordance with the rules and constraints of a well-defined Computational Model.

The Computational Model which we have elected to underpin our engineering approach is based on the deadline monotonic scheduling theory, as established by [Audsley et al., 1991, Audsley et al., 1993]. Section 3.4.5 outlined the key elements of this theory.

Systems built in conformance with the deadline monotonic scheduling theory lend themselves to static response time analysis. Systems designed with the HRT-HOOD method possess this
property by construction. The basis to response time analysis as an advanced form of static timing analysis were first laid down by [Joseph and Pandia, 1986] and subsequently furthered by [Audsley et al., 1991]. The theoretical foundations of our approach to response time analysis were presented in section 4.5.3 (and, in a more concise form, in [Vardanega, 1996]). The analysis model stipulates that the response time of one thread be defined as the longest elapsed time it takes for that thread to complete its most demanding set of processing activities in response to an activation occurring under maximum contention from the rest of the system. Threads in the system are schedulable if and only if they can fulfill their real-time commitments, expressed in terms of activation frequency and deadline, under those initial conditions. The example presented in the following shows how, in our model, the development process takes these notions into account and progressively consolidates across the four steps introduced in section 4.3.1.

While the structure of the chapter reflects the same global progression, the presentation will also show that the overall development process entails the orderly execution of the following finer-grained activities:

1. use of the HRT-HOOD objects (i.e. cyclic, sporadic, protected, passive) to establish the initial logical model of the system;
2. determination of the real-time requirements associated with the operation of those objects;
3. construction of the operational profiles associated with the processing activity required of those objects (e.g.: use and sequence of internal and / or external execution requests);
4. automated generation of the program structure corresponding to the system in terms of the HRT-HOOD Computational Model and provision of the functional code associated with the object operations;
5. automated extraction of the worst-case execution time profile for all objects in the system;
6. static analysis of the real-time operation of the system and interpretation of the results and determination of the feedback information required for the completion / consolidation of the development process.

The intended objective of each of the main development steps discussed in the following is recalled at the beginning of the corresponding section.

5.3 Step B.1: Real-Time Design

Objective: the design of the system is established to the desired level of detail by use of the HRT-HOOD design method.

Let us assume we want to implement (a portion of) a real-time system comprised of: (a) an entity in charge of producing data at a fixed rate (say, 20 milliseconds) in a rigidly regular order;
and (b) an entity in charge of consuming those data on request from the Producer. Let us also assume that the Producer and the Consumer have to place a time-stamped record of the correct progression of their activity into a dedicated data structure and that the system is also to include: (c) an entity in charge of transmitting the contents of that data structure to the system operator also on request from the Producer. Let us finally assume that all of these entities also have to perform a number of other auxiliary processing activities as part of every nominal activation.

We now want to describe these entities and their interactions in terms of HRT-HOOD objects and relations. Moreover, we want to model the three main entities of our system (the Producer, the Consumer and the transmitter) as distinct concurrent activities. To better illustrate our point in this chapter, though, we will do so using as many of the HRT-HOOD objects as the example can accommodate, in preference to seeking design and/or implementation optimisation.

We know from section 4.4 that HRT-HOOD objects may be cyclic, sporadic, protected or passive. We also know that concurrent (i.e. active) objects may only be either cyclic or sporadic. In our case, Producer clearly maps to a cyclic object, while the operation of Consumer and the transmitter (which we will call Print_Tool) is tied to an activation trigger coming from Producer and, hence, can be regarded as being software sporadic. The data generated by Producer will be stored in a dedicated protected object (which we will call Buffer) from which they will be retrieved by Consumer. Similarly, Producer and Consumer will place their time-stamped report into another dedicated protected object (which we will call Store). Print_Tool will extract from Store the information to be sent to the system operator. Finally, we will gather the auxiliary processing activity to be performed by Producer and Consumer into a passive object called Background, while we will place the auxiliary activity of Print_Tool into another passive object named PrintFct.

Figure 5.1 shows our HRT-HOOD system as a collection of co-operating terminal objects. The overall system is denoted by one single cyclic parent object named System_Main, which includes:

- one cyclic terminal object named **Producer**;
- one protected terminal object, named **Buffer**, which provides two protected synchronous operations: Read and Write;
- one software sporadic terminal object named **Consumer**;
- another software sporadic terminal object named **Print_Tool**;
- two passive terminal objects, named **Background** and **PrintFct**, each providing one unconstrained operation (Work and Output_Text respectively).

The System_Main parent object is an active object and hence allows for internal independent flow of control. This is allocated to all of the independent threads of control which the parent object ultimately decomposes to, that is: one cyclic object (Producer) two software sporadic objects (Consumer and Print_Tool) and one protected object (Buffer) which globally caters for a total of three threads, two OBCS (synchronisation servers) and one resource server.

The HRT-HOOD type of each object in the system is denoted by the letter tag placed at the top left of the object icon: the C tag denotes a cyclic object; S denotes a sporadic object; Pr denotes
a protected object. A no-tag banner denotes passive objects. Software sporadic objects differentiate from interrupt sporadic objects for the type of the respective triggering event: an ASER asynchronous trigger would denote a software sporadic; conversely, an ASER BY IT would denote an interrupt sporadic (the latter not being used in the example system).

The boxes placed to the right of the System Main parent object denote further objects which are used by child objects of System Main but are defined externally to it.

The E tag placed to the left of the object box denotes that the external object in question is defined as an environment object (i.e. a system-level library). Environment objects are typically used to provide for primitive data definitions (e.g.: types, constants, variables) of general interest to the application.

In particular, the RTA environment object is always required by any HRT-HOOD system for it provides the base definition of all the real-time attributes needed by the real-time objects in the system.

Furthermore, as every HRT-HOOD system may be used as a subsystem component of a higher-level system, external objects may also be of any other HRT-HOOD base type and rep-
resent child objects of an external subsystem which are made visible to the child objects of the local subsystem. This is, for example, the case with the protected object \textbf{Store}, which will be used – but not exclusively – by child objects of System..Main.

HRT-HOOD objects may provide external operations (execution requests) for other objects to invoke. The cyclic terminal object Producer in the example system shown in figure 5.1 provides no external operation. Conversely, the external interface of both Consumer and Print..Tool, each of which is a software sporadic terminal object, provides the \textit{Start} operation required to trigger the sporadic operation of the object. As required by the method, both instances of \textit{Start} operation are asynchronous, as denoted by the ASER tag placed next to the operation symbol. The protected terminal object Buffer provides for two protected synchronous operations (PSER) \textit{Read} and \textit{Write}. The example system also includes two passive terminal objects each of which provides an unconstrained operation.

The overall operation of the system is governed by the periodic execution of Producer (which explains why the parent object System..Main is marked as a cyclic object). At every periodic activation, Producer:

- deposits a data item in the Buffer using the \textit{Write} operation of the latter;
- performs a given amount of internal work using the \textit{Work} operation provided by the Background object;
- records in \textit{Store}, using the \textit{Write} operation, the time-stamped mark of completion of the current operation;
- activates Consumer and Print..Tool which shall, respectively, retrieve the data item produced by Producer and report to the output port on the current status of the system operation using the activity record left by Producer and Consumer in \textit{Store}.

The arrows between objects in the diagram denote the "use" relationships in place between any two objects in the system. In our case, the arrows departing from the Producer object denote that Producer uses:

- the protected synchronous \textit{Write} operation provided by the Buffer protected object;
- the unconstrained \textit{Work} operation provided by the Background passive object;
- the asynchronous \textit{Start} operation provided by the Consumer sporadic object;
- the protected synchronous \textit{Write} operation provided by the \textit{Store} protected object (not visible in figure 5.1);
- the asynchronous \textit{Start} operation provided by the Print..Tool sporadic object.

Producer, Consumer and Print..Tool are \textit{active} objects, hence possess independent threads of control. The operation of each such thread of control is subject to specific real-time requirements,
which are supplied with the definition of the relevant object. All the active objects in our system will be mission critical.

Let us now define the real-time requirements placed on the overall operation of the system: (a) Producer would have a period of 20 milliseconds and a deadline of 9 milliseconds, so that it can complete an activation before Consumer starts its; (b) Consumer would have a minimum interarrival time strongly correlated with the period of Producer and a deadline placed slightly earlier than the start of the next 20-millisecond cycle of Producer, say at 17 milliseconds, so as to guarantee a tolerance margin for other activities in the system; (c) Print_Tool would have the same minimum interarrival time as Consumer but a deadline shorter than Producer, to denote the criticality of its timely reporting to the user.

HRT-HOOD allows the designer to structure and formalise these real-time requirements at the level of the Object Description Skeleton (ODS). The main requirements and design attributes described in the ODS can be divided in two categories as follows: (1) those which are to be defined directly by the designer as derived from the user specification; these are period, deadline and criticality; and (2) those which directly result from the rules of the scheduling theory underpinning the Computational Model of choice; this is the case, for example, of the priority attribute, which results from application of the priority assignment algorithm embodied in the proposed toolset concept and outlined in section 5.5. According to the convention adopted in the proposed toolset concept, the criticality level of a real-time active object of type cyclic and sporadic may be HARD, SOFT or NON_CRITICAL to denote the highest, medium and lowest level, respectively.

Object Producer Is
Pragma Target_language(NAME => Hrt_language);
Cyclic
Description
Real_time_attributes
  Period
    Mode_1 => 0.020
Deadline
    Mode_1 => 0.009
Priority
    6
Importance
    Mode_1 => hard
...
End_object Producer

Figure 5.2: HRT-HOOD Object Description Skeleton Fragment (Producer).

Figure 5.2 shows a fragment of the ODS of Producer, where the cyclic operation of the object is assigned (among other) the following real-time attributes:

- a period of 0.020 seconds;
- a deadline of 0.009 seconds;
• a hard criticality;

• a priority level of 6 (which assumes that the first iteration of development step B.3(a) – where priority assignment is performed – has already been carried out).

In principle, the attributes set in figure 5.2 are mode dependent, as the actual operation of the system (and, consequently, its real-time requirements) may in fact be mode dependent. In our example, however, we will not address issues of mode change.

In addition to the object’s real-time requirements, the ODS is also used to specify exactly what external operations the object requires from other objects in the system. This serves to qualify the relationship at object level denoted by the arrows in the design diagram.

5.4 Step B.2: Binding to Computational Model

Objective: the Ada program corresponding to the current HRT-HOOD design is automatically extracted from the design tool.

We have seen in section 4.5.1 that the HRT-HOOD Computational Model precisely defines the Ada entities, constructs and implementation which correspond to the objects and relations supported by the HRT-HOOD design method. This definition extends the general rules for the extraction of Ada source files from HOOD design which are specified by the HOOD standard [HTG, 1993] and permits the generation of a complete source Ada program from an HRT-HOOD design. We formalised this definition as part of the work performed for the implementation of the technology described in section 4.5. [Intecs Sistemi, 1996b] and [Intecs Sistemi, 1997] describe the Ada code extraction rules required to map an HRT-HOOD design to a concurrent Ada program compliant with the HRT-HOOD Computational Model. [Intecs Sistemi, 1996a] is an industrial-quality implementation of an HRT-HOOD design tool which embodies a code extractor governed by those rules. Thanks to these code extraction rules, the generation of the Ada program from an HRT-HOOD design becomes a fairly straightforward exercise. The whole of the program structure and a large proportion of other code components, in fact, directly emanate from the attributes of the design objects and the relationships between them. In the following, we will briefly illustrate how the code generation process works and show fragments of the Ada code which correspond to the implementation of the example system introduced in the previous section.

All of the code fragments shown in figures 5.3 to 5.11 in the following are displayed exactly as produced by our HRT-HOOD code extractor tool. In this respect, we must stress that our emphasis here is placed on the illustration and demonstration of the concept rather than on the optimisation of the generated code.

The interface and operation(s) of an HRT-HOOD object naturally map to the specification and the body, respectively, of the Ada package destined to represent it. The object interface must be visible and accessible to other objects and, therefore, maps to the specification of the corresponding package. The internal operations of the object map to the package body. The "use" relationship between objects at design level maps to the "with" relationship between the
corresponding Ada packages. This is reflected, for example, by figure 5.3, which shows the
"with" preamble of the package corresponding to the Producer object.

```ada
with Real_Time; use Real_Time;
   -- REQUIRED OBJECT : Buffer |-- OPERATION(S) : Write
with Buffer;
   -- REQUIRED OBJECT : Print_Tool |-- OPERATION(S) : Start
with Print_Tool;
   -- REQUIRED OBJECT : Background |-- OPERATION(S) : Work
with Background;
   -- REQUIRED OBJECT : Consumer |-- OPERATION(S) : Start
with Consumer;
   -- REQUIRED OBJECT : Store |-- OPERATION(S) : Write
with Store;

package body Producer is -- CYCLIC object ...
```

Figure 5.3: "With" Preamble of Producer (body).

- The real-time attributes defined in the ODS of Producer (cf. figure 5.2) are directly trans-
lated to the code of the `Producer_RTATT` package (a fragment of which is shown in figure
5.4); one such package is created for every individual cyclic, sporadic and protected termi-
nal object in the system (passive objects have no real-time properties).

```ada
with System; use System;
with RTA; use RTA;
with Real_Time; use Real_Time;
package Producer_RTATT is
   THREAD_PERIOD : constant MODE_DURATION :=
      (Mode_1 => Real_Time.To_Time_Span (0.020));
   THREAD_DEADLINE : constant MODE_DURATION :=
      (Mode_1 => Real_Time.To_Time_Span (0.009));
   THREAD_IMPORTANCE : constant MODE_IMPORTANCE :=
      (Mode_1 => hard);
   THREAD_PRIORITY : constant PRIORITY := 6;
   ...
end Producer_RTATT;
```

Figure 5.4: Producer Real-Time Attributes (Fragment).

- The periodic operation of Producer is modelled by the periodic task THREAD shown in
figure 5.6 and is based on the predefined cyclic task template discussed in section 4.5.
The code structure of THREAD is *completely standard*, for the sole specific customisation required to its operation are fully encapsulated in the procedure `Thread_Action` shown in figure 5.5 and in the INTERNAL DATA section of the package shown in figure 5.6; both units are based on the definitions and code components supplied by the user in the ODS of the object.

```pascal
procedure Thread_Action is
begin
    Buffer.Obsc.Write (Item);
    Background.Work (Workload);
    Consumer.Obsc.Start;
    Store.Obsc.Write (Item, 1, Real_Time.Clock);
    Print_Tool.Obsc.Start;
end Thread_Action;
```

**Figure 5.5: Operation of Producer (separate body).**

The code corresponding to the sporadic object Consumer is generated almost exactly as described for Producer. The only notable difference between the two lies in that Consumer is a *software sporadic* object and, hence, it needs an OBCS to command its activation on arrival of the triggering event (which, in this case, is generated by Producer).

- The OBCS of Consumer must be visible to the object which supplies the required trigger (i.e. Producer). The synchronisation server which implements the OBCS must be therefore declared in the specification of the Consumer package (cf. figure 5.7) so that the latter may be "with"-ed by the body of the Producer package.

- The OBCS of Consumer is implemented as a classical Ada 83 passive task, as shown in figure 5.8. The mapping model translates directly to an implementation based on an Ada 95 protected object with the `WAIT_Start` operation coded as a protected entry and `Start` coded as a protected procedure.

    The same way the code generator encapsulates the periodic operation of cyclic objects within a procedure named `Thread_Action`, the operation of sporadic objects maps to a procedure named `OPCS_Start`, as shown in figure 5.9.

    The sporadic operation of Consumer shown in figure 5.9, which the User describes in the code section of the ODS of the object, consists of reading out from Buffer the information item previously stored by Producer, recording in Store the successful completion of the action and performing a certain amount of internal work before returning control until arrival of the next software trigger.

    Whereas the OBCS of software sporadic objects map to a *synchronisation server* task but do not represent an HRT-HOOD object on their own, *resource server* tasks directly model the HRT-HOOD protected object. Buffer is a protected object and, hence, maps to a resource server.
package body Producer is -- CYCLIC object

--------------- INTERNAL OPERATION(S) ---------------
procedure Thread_Action;
--------------- INTERNAL DATA ---------------
Workload : constant Integer := 500;
Item : Integer := 0;

task THREAD is
  pragma PRIORITY( Producer_RTATT.THREAD_PRIORITY);
end THREAD; -- from HRT attributes

task body THREAD is
  T : Real_Time.TIME := RTA.SYSTEM_START_UP_TIME;
begin
  loop
    Real_Time.delay_until(T);
    Thread_Action;
    T := T + Time_Span(Producer_RTATT.THREAD_PERIOD(RTA.CURRENT_MODE));
  end loop;
end THREAD;

--------------- OPCS OF UNCONSTRAINED OPERATIONS ---------------
procedure Thread_Action is separate;
end Producer;

Figure 5.6: Producer (body).

in which every protected synchronous operation (PSER) declared in the external interface of the object maps to one dedicated protected entry provided with IN and OUT parameters. The specification of the Buffer object is shown in figure 5.11.

5.5 Step B.3(a): Extraction of Design Requirements

Objective: a descriptive representation of the real-time requirements established on the system is automatically constructed from the real-time attributes of the current HRT-LOOP design.

As mentioned in section 5.2, the most prominent feature of an HRT-LOOP design is that it lends itself, by construction, to response time analysis, which is the form of static timing analysis adopted as part of our engineering concept.

In order to allow for response time analysis to be iteratively performed along with the progress of the development, the toolset must be able to automate the extraction of the information required for the analysis from all of the time-critical components of the system.

We know from the discussion in section 4.5.3 that response time analysis determines whether
with SYSTEM; use SYSTEM;
with RTA; use RTA;
with Real_Time; use Real_Time;
with Consumer_RTATT;
package Consumer is -- SPORADIC object
---------- PROVIDED INTERFACE ----------
type Start_PARAMETER_SET is record
  OVERRUN : BOOLEAN := FALSE;
  Start_TIME : Real_Time.TIME;
end record;
---------- OBJECT CONTROL STRUCTURE ----------
task OBCS is
  pragma PRIORITY(Consumer_RTATT.INITIAL_CEILING);
  -- from HRT attributes Consumer
  pragma Passive;
  entry Start;
  entry WAIT_Start(THE_PARAMS : out Start_PARAMETER_SET);
  -- called by THREAD task only
end OBCS;
end Consumer;

Figure 5.7: Consumer (spec).

every individual thread in the system can meet its real-time requirements under maximum con-
tention from the rest of the system. Response time analysis determines the worst-case completion
time of a thread as a function of the following components:

1. The worst-case processing requirement of that thread, WCCT), which results from the
worst-case execution time profile generated from the Ada program corresponding to the
HRT-HOOD design in the fashion described in section 4.5.3. The worst-case execution
profile generated for our example system is shown in figure 5.13 and discussed in section
5.6.

2. The B factor, which is determined as the largest value between the single longest period of
defered preemption potentially incurred at run-time time and the longest protected service
performed by a higher-ceiling server for a lower-priority thread. The former is a static
property of the run-time environment; the latter depends on the "use" relationships between
objects in the design and the chosen priority assignment.

3. The I factor, which is a function of the priority levels assigned to the threads in the system
and, hence, of the overall system load.

The HRT-HOOD Computational Model assumes the use of priority-based preemptive schedul-
ing. Hence, threads have to be assigned the designated priority level before they can be subject
to response time analysis. With our approach, this assignment can be performed very early in
the development process and successively reconfirmed as often as desired. This function operates
task body OBCS is -- Consumer.OBCS
begin
loop
select
accept Start do
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;

Figure 5.8: Consumer OBCS (body).

upon a textual representation of the user-defined design properties which effect the assignment. These properties include:

- the criticality and deadline of all threads in the system (which are part of the threads' real-time commitments), for the relative ordering of the priority assigned to threads;

- the "used by" profile of all servers in the system (which is a static property of every HOOD design component), for the determination of the ceiling priority which they have to be assigned.

Both complements of information can be statically determined from explicit properties of the design. The former is extracted from the ODS of all cyclic and sporadic terminal objects and recorded in a file called the User Configuration File (a fragment of which is shown in figure 5.12). The latter can be obtained in the two following distinct ways: (1) from static analysis of the "use" relationships between objects in the design (this generation path uses the same principle which provides for the determination of the "with" preamble of an object like the one shown in figure 5.3); (2) from WCET processing of the Ada program automatically generated from the current design.

Technique (1) is very convenient when the system design is merely sketched and contains no actual code other than the program structure. Conversely, technique (2) becomes very practical
procedure OPCS_Start(OVERRUN : in BOOLEAN;
   Start_TIME : in TIME) is
begin
  Buffer.Obsc.Read(Item);
  Store.Obsc.Write (Item, 2, Real_Time.Clock);
  Background.Work (Workload);
end OPCS_Start;

Figure 5.9: Operation of Consumer (separate body).

as soon as the system starts having some representative code components in it. Whatever the
generation technique, though, the resulting information is recorded in a textual file called the
Execution Skeleton File, a fragment of which is shown in figure 5.13.

The priority assignment algorithm requires that tasks must not share the same priority level.
This prescription ensures the same run-time scheduling behaviour in the face of implementation-
specific features of the run-time environment of choice. Tasks with decreasing criticality are
assigned decreasing priority levels. Tasks within the same criticality range are assigned priority
levels in deadline monotonic fashion. Server tasks are assigned a ceiling priority level which
is set at least one level higher than the maximum priority of their client tasks. As a result of
this scheme, the priority of all of the real-time active (terminal) objects in the system is readily
determined upon definition of the user-defined design attributes established in the ODS of the
relevant objects.

The priority assignment function is embedded in both the Scheduling Analysis and Scheduler
Simulator tools shown in figure 4.6.

5.6 Step B.3(b): Generation of WCET Profile

Objective : the worst-case execution (WCET) profile of the system is automatically generated
from the Ada program extracted from the current HRT-HOOD system.

The primary means for the generation of the WCET profile of the application to be recorded
in the Execution Skeleton File is via the capability built in the Ada Compilation System.

The WCET profile generation capability, which operates in accordance with the principles
discussed in section 4.5.3, uses a stylised syntax to describe the worst-case execution profile of all
threads and servers in the system.

Threads are described as follows:

THREAD <Ada_Name>  -- full Ada name of thread
TYPE [CYCLIC | SPORADIC | INTERRUPT SPORADIC] -- thread type
-- worst-case execution profile expressed as a suitable
-- combination of the following statements
WCET Cp, Cmr, Cmw -- timing of basic block expressed as a triplet
-- of values for the corresponding amount of
package body Consumer is -- SPORADIC object

-------- INTERNAL DATA --------

Workload : constant Integer := 500;
Item : Integer := 0;
-------- INTERNAL OPERATION(S) --------

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

Start_PARAMETERS : Start_PARAMETER_SET;
Start_CALLED : BOOLEAN := FALSE;
task THREAD is
pragma PRIORITY (Consumer.PERIOD.INITIAL_PRIORITY);
end THREAD; -- from HRT attributes

task body THREAD is
Start_BUFFER : Start_PARAMETER_SET;
beg

loop

OBCS.WAIT_Start(Start_BUFFER);
OPCS_Start(Start_BUFFER.OVERRUN,Start_BUFFER.Start_TIME);
end loop;
end THREAD;

-- task body OBCS shown in figure 5.8
-------- OPCS OF CONSTRAINED OPERATIONS --------

--|:OPCS_CODE <Start> shown in in figure 5.9

end Consumer;

Figure 5.10: Consumer (body).

-- processing cycles (Cp) memory read cycles (Cmr)
-- and memory write cycles (Cmw)

CALL_PO <Server_Ada_Name> <Entry_Name> -- for every server call
LOOP <Integer_Count> -- for every bounded loop in the profile
 <Loop_Body> -- as a combination of WCET, CALL_PO and LOOP statements
END

-- list of protected calls which have been excluded from the
-- worst-case execution profile
PO <PO_Name> <Entry_Name>
END <Ada_Name>

The syntax used to describe servers varies slightly with their type with respect to our Compu-
tational Model. Resource servers are described as follows:

PROTECTED <Server_Ada_Name>

TYPE RESOURCE
ENTRY <Entry_Name_i> -- for every protected entry
-- worst-case execution profile of entry expressed using the
-- syntax shown for the execution profile of threads
-- list of protected calls which have been excluded from the
-- worst-case execution profile of the entry
PO <PO_Name> <Entry_Name>
END <Server_Ada_Name>
with SYSTEM; use SYSTEM;
with RTA; use RTA;
with Buffer_RTATT;
package Buffer is -- PROTECTED object
-------------- OBJECT DESCRIPTION --------------
Minimum : constant := 0;
Maximum : constant := 100;
subtype Buffersize is integer range Minimum .. Maximum;
------------- OBJECT CONTROL STRUCTURE -------------
--- CONstrained OPERATION(S) ---
-- Write constrained by PSER
-- Read constrained by PSER
task OBOS is
  pragma PRIORITY(Buffer_RTATT.INITIAL_CEILING);
    -- from HRT attributes Buffer
  pragma Passive;
  entry Write(Item : in Integer);
  entry Read(Item : out Integer);
end OBOS;
end Buffer;

Figure 5.11: Buffer (spec).

Synchronisation servers (i.e. OBOS of software sporadic objects) are described as follows:

PROTECTED <Server_Ada_Name>
TYPE SYNCHRO
-- OBOS have one barred entry
BARRIER WCET Cpc, Cmr, Cmw -- worst-case time for the
                       -- evaluation of the barrier
ENTRY <Barriered_Entry_Name> -- only one
-- worst-case execution profile of barred entry
ENTRY <Unbarriered_Entry_Name_i> -- for every other
                       -- unbarriered entry
-- worst-case execution profile of unbarriered entry
-- list of protected calls which have been excluded from the
-- worst-case execution profile of the entry
PO <PO_Name> <Entry_Name>
END <Server_Ada_Name>

All profiles are basically comprised of the three following distinct elements of information:

- The thread or server type, which the analysis tools use to verify that the corresponding execution profile complies with the properties expected of the object.

- The worst-case execution profile determined by the built-in capability of the compilation system, which the analysis tools use to compute the WCCT component of the thread’s response time; in order to allow for the order-sensitive scheduling simulation supported by
THREAD DEFINITION
THREAD Consumer.THREAD
  CRITICALITY hard
  MINIMUM 200000
  DEADLINE 170000
END -- Consumer

THREAD Producer.THREAD
  CRITICALITY hard
  PERIOD 200000
  DEADLINE 90000
END -- Producer
END

Figure 5.12: Application User Configuration File (Fragment).

the proposed toolset, the worst-case execution profile generation maintains the canonical order of all of the relevant execution components.

• The list of protected calls possibly made by the thread outside the worst-case execution profile; this information is needed to correctly reflect the use relationship of threads to servers irrespective of the retained execution profile and accordingly assign the appropriate ceiling priority to servers; this information is also used to flag the occurrence of cases in which the blocking incurred by a thread from access to shared servers occurring outside the worst-case execution profile is greater than that computed from the retained profile (we discussed this event in section 4.5.3).

On the whole, the worst-case execution profile of a thread is comprised of sequential blocks of execution (each denoted by a WCET entry) delimited by explicit server calls and or start of bounded loops. All the \( < C_p, C_{mr}, C_{mw} > \) values specified in WCET entries of a thread’s execution profile are determined by summation of the execution cost of the assembly instructions enclosed within the boundaries of the relevant source block and exclusively belonging to the thread’s own code. The execution cost of the individual assembly instructions is specified in a so-called Target Characteristics File for the chosen target board configuration. One thread’s WCCT is, thus, computed by summation of all the WCET components and server calls included in the thread’s profile plus the cost of all the run-time system services required for management and administration support of the thread’s operation.

A fragment of the WCET profile generated for the Ada program extracted from the HRT-HOOD system shown in figure 5.1 is reported in figure 5.13. The reader is invited to relate the execution profile of the thread of control of Producer with the call profile of the object as outlined in section 5.3 and the code profile of the procedure Thread_Action shown in figure 5.5. Similarly, the execution profile of the thread of control of Consumer may be easily related to the main-loop body of the thread shown in figure 5.7 in conjunction with the body of the procedure OPCS_Start shown in figure 5.9.
PROGRAM SYSTEM_MAIN
THREA D PRO DUC ER.THREAD
TYPE CYCLIC
C ALL PO BUFFER.OBCS WRITE
WCET 5 0570, 12614, 4 211
C ALL PO CONSUMER.OBCS START
C ALL PO STORE.OBCS WRITE
C ALL PO PR I NT_TOOL.OBCS START
END
THREA D CONSUMER.THREAD
TYPE SPORADIC
C ALL PO CONSUMER.OBCS WAIT_START
C ALL PO BUFFER.OBCS READ
C ALL PO STORE.OBCS WRITE
WCET 50490, 12606, 4210
END
PRO TECTED CONSUMER.OBCS
TYPE SYNCHRON
ENTRY START WCET 35, 2, 1
BARRIER WCET 6, 1, 0
ENTRY WAIT_START WCET 38, 5, 5
END
PRO TECTED BUFFER.OBCS
TYPE RESO URCE
ENTRY READ WCET 89, 9, 5
ENTRY WRITE WCET 90, 8, 6
END -- ...
END

Figure 5.13: Application Execution Skeleton File (Fragment).

5.7 Step B.4: Feedback to Design

Objective: the data extracted from the current design of the system are fed to the static analysis tools included in the proposed toolset so as to obtain prediction, confirmation and verification or otherwise of the correct timing behaviour of the system and to amend and/or consolidate the design of the system accordingly.

The key aim of introducing steps B.2-B.3(b) as easily-achieved enhancements to the development process is to provide direct support the iterative and incremental nature of real-time embedded software development discussed in section 4.3.2. This allows the designer to analyse the timing behaviour of the system at virtually all stages of development and as early as from the establishment of the logical model. The attributes of the system which are relevant to this analysis include the real-time requirements established on the system (which are collected in the User Configuration File generated in step B.3(a)) and the worst-case execution profile of all threads in the system (which is recorded in the Execution Skeleton File generated in step B.3(b)).

Our timing analysis tools perform schedulability analysis and scheduling simulation on the
model of the system represented by the data files extracted from the current HRT-HOOD design. The two forms of analysis serve complementary purposes. The former performs the classical worst-case response time analysis and determines whether the current system is capable of fulfilling the assigned real-time requirements; the latter predicts the sequence of scheduling events that the current structure of the application should encounter at run-time under a user-defined scenario (defining, for example, the desired pattern of arrival of external interrupts). The schedulability analyser performs a coarse analysis of the system's ability to meet its real-time commitments. The scheduler simulator provides the user with a finer-grained insight into the way in which the execution of the system (described in terms of the succeeding scheduling events) progresses within selected time intervals. Both tools operate on the same input base. This includes:

- the User Configuration File and Execution Skeleton File discussed in section 5.6; and
- the Run-Time Characteristics File, which describes the worst-case overhead to be encountered at run-time from the execution of the (restricted set of) thread management services utilised by the application. We have shown in section 4.5.2 the timing characterisation of our Ada run-time implementation.

Table 5.1 shows a fragment of the report generated by the scheduling analysis tool. The report includes the following information items (all time figures are expressed in microseconds):

- the summary of the user-defined real-time attributes attached to the individual threads of control (namely: criticality; period or minimum interarrival time; deadline)
- the summary of the computed real-time properties determined by response time analysis for the individual threads (namely: priority； WCCT — computed converting the cycle count information obtained for the thread from its worst-case execution profile to the board frequency and memory read and write wait states set for the target board under consideration in the board-specific part of the User Configuration File —； response time and schedulability status; blocking time and blocking cause)
- the indication of the sensitivity of the thread's WCCT to load increase / decrease (expressed in percent of the current WCCT value for that thread) which would make (or keep) the entire thread set fully schedulable
- the list of servers used by the application with indication of the respective ceiling priority and the relevant list of client threads
- the indication of the current level of worst-case CPU utilisation.

Threads and servers are identified in the report by the numeric identifier assigned according to the respective lexical order of declaration in the User Configuration File. For example, the thread of Consumer (Consumer . Thread) is identified in table 5.1 as thread #1; the thread of Producer (Producer . Thread) is identified as thread #2; the OBCS of Consumer (Consumer . OBCS)
is identified as protected object #5; and the Buffer resource server (Buffer.OBCS) is identified as protected object #4 (servers being numbered after their lexical order of declaration in the Execution Skeleton File).

Amongst other things, table 5.1 shows that all threads in the example system are schedulable and incur worst-case blocking caused by the run-time system (as an indication that the adverse effect of blocking caused from use of IPCI protection of shared servers is presently negligible).

Furthermore, the margin analysis data shown in the rightmost column of table 5.1 inform the user that, for example, the WCCT of Producer (thread #2) of can still be increased by up to 22.9% without effecting the schedulability status of the entire system. It can easily be noticed, in fact, that, upon such an increase, the response time of Producer_Thread would still be within about 100 µs from the relevant deadline (8.892 vs 9.000) while the overall CPU load of the system would rise up to 83.22%, from the level of 75.21% reported in figure 5.1, without this causing any threads to miss their deadline.

**Table 5.1: Scheduling Analysis Report (Fragment).**

<table>
<thead>
<tr>
<th>Sch</th>
<th>Th#</th>
<th>Crit</th>
<th>Deadline</th>
<th>Prio</th>
<th>WCCT</th>
<th>Period</th>
<th>Response</th>
<th>Blocking</th>
<th>Margin</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>3</td>
<td>HARD</td>
<td>8000</td>
<td>9</td>
<td>160</td>
<td>20000</td>
<td>428</td>
<td>130</td>
<td>1.00E+03</td>
</tr>
<tr>
<td>yes</td>
<td>2</td>
<td>HARD</td>
<td>9000</td>
<td>6</td>
<td>6999</td>
<td>20000</td>
<td>7394</td>
<td>130</td>
<td>2.29E+01</td>
</tr>
<tr>
<td>yes</td>
<td>1</td>
<td>HARD</td>
<td>17000</td>
<td>5</td>
<td>6875</td>
<td>20000</td>
<td>14341</td>
<td>130</td>
<td>3.87E+01</td>
</tr>
<tr>
<td>yes</td>
<td>5</td>
<td>HARD</td>
<td>18000</td>
<td>2</td>
<td>213</td>
<td>40000</td>
<td>14521</td>
<td>130</td>
<td>1.63E+03</td>
</tr>
<tr>
<td>yes</td>
<td>4</td>
<td>HARD</td>
<td>34000</td>
<td>1</td>
<td>142</td>
<td>40000</td>
<td>14735</td>
<td>130</td>
<td>3.63E+03</td>
</tr>
</tbody>
</table>

List of all protected objects:

<table>
<thead>
<tr>
<th>PO#</th>
<th>Pr</th>
<th>Protected Type</th>
<th>PO User List</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>RESOURCE</td>
<td>T#2, T#1,</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>SYNCHRONISATION</td>
<td>T#2, T#1,</td>
</tr>
</tbody>
</table>

Thread Set Utilisation Factor: 7.52133E-01

The sensitivity information returned by this type of analysis represents a powerful means for the user to determine the feasibility and the limits of pre- or post-launch modifications to the structure and operation of (components of) the system.

Analysis data to the same effect can also be obtained from the scheduling simulator tool. Table 5.2 shows, in fact, a fragment from the output report of the tool which provide a statistical overview of selected aspects of the execution of the system.

Among other things, table 5.2 provides the following elements of information:

- the overall execution time of every thread in the system over the user-defined simulated execution time (about 6 seconds in this case);
- the corresponding amount and relative proportion of run-time system overhead incurred over the observed execution;

- the minimum and maximum margin from all the deadlines achieved (or to any missed deadline) during the observed execution.

Interestingly, data reported in table 5.2 can be used to confirm and qualify predictions obtained from the worst-case analysis performed by the schedulability analysis tool. For example, we can observe that the CPU load (inclusive of the relevant run-time system overhead component, which amounts to some 3.5%) measured by the simulator over a realistic user scenario is fairly close to the worst-case CPU load predicted by the analyser (70.94 vs 75.21). This indicates that the operation of the example system is rather steady. In fact, this is not really surprising, for the simple system in the example receives no external interrupts and all the sporadic activities in the reference scenario are triggered at fixed intervals by fixed-period cyclic activities. In this case, therefore, the load predicted by the simulator is bound to be inferior to the prediction provided by the analyser.

Table 5.2: Fragment of Scheduler Simulator Output (Statistics).

<table>
<thead>
<tr>
<th>THREAD</th>
<th>CPU TIME</th>
<th>RTS</th>
<th>ACHIEVED DEADLINE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SUM</td>
<td>%</td>
<td>SUM</td>
</tr>
<tr>
<td>CONSUMER.THREAD</td>
<td>2019928</td>
<td>33.70%</td>
<td>42861</td>
</tr>
<tr>
<td>PRODUCER.THREAD</td>
<td>2005555</td>
<td>33.46%</td>
<td>88392</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simulation_Time : 5994128 (usec)
Processing_Time : 4252489
Overall CPU : 70.94 %
Overall RTS : 3.52 %

The situation may significantly differ in the more typical situation whereby the system includes sporadic activities triggered by external events. In this case, in fact, the analysis is performed on the basis of the design assumptions which bound the interarrival time of the external interrupts, whereas the simulation may be run upon statistical arrival patterns. Under these circumstances, the results from the simulation may reveal deficiencies in the design assumptions used for the analysis. Such deficiencies may, for example, take the form of an activity missing a deadline in the presence of an interrupt burst not contemplated by the analysis assumptions or a significantly higher CPU load under the same condition. These events suggest the need to revisit the requirements and assumptions of the system and possibly iterate the analysis cycle.

To reinforce the argument that the results from the simulation may be used to confirm and qualify the predictions of the analysis, it can be noticed that table 5.2 also shows that Producer.-Thread consistently completes its activations within a margin of about 20% to its deadline.
This reads very much in line with the data from the margin analysis reported in table 5.1 which indicated for that thread the potential for a 22.9% increase in its WCCT component.

In addition to providing the statistics information shown in table 5.2, the scheduler simulator also predicts the sequence of scheduling events incurred by the application at run time under a given execution scenario and constructs for the user a log of the events occurring within a selected window of observation. The ability to construct the execution scenario for which to obtain (and visualise) the predicted scheduling of run-time events is an extra aid for the designer to build confidence in the expected operation and performance of the system.

Table 5.3: Fragment of Scheduler Simulator Output (Schedule).

<table>
<thead>
<tr>
<th>TIME</th>
<th>EVENT</th>
<th>THREAD</th>
<th>PO</th>
<th>PRIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>6865</td>
<td>ENTER_PROTECTED</td>
<td>PRODUCER_THREAD</td>
<td>CONSUMER_OBCS</td>
<td>6</td>
</tr>
<tr>
<td>6888</td>
<td>LEAVE_PROTECTED</td>
<td>PRODUCER_THREAD</td>
<td>CONSUMER_OBCS</td>
<td>7</td>
</tr>
<tr>
<td>6888</td>
<td>LOWER_BARRIER</td>
<td>PRODUCER_THREAD</td>
<td>CONSUMER_OBCS</td>
<td>7</td>
</tr>
<tr>
<td>6888</td>
<td>ENTER_PROTECTED</td>
<td>PRODUCER_THREAD</td>
<td>STORE_OBCS</td>
<td>6</td>
</tr>
<tr>
<td>6925</td>
<td>LEAVE_PROTECTED</td>
<td>PRODUCER_THREAD</td>
<td>STORE_OBCS</td>
<td>11</td>
</tr>
<tr>
<td>6925</td>
<td>ENTER_PROTECTED</td>
<td>PRODUCER_THREAD</td>
<td>PRINT_TOOL_OBCS</td>
<td>6</td>
</tr>
<tr>
<td>6948</td>
<td>LEAVE_PROTECTED</td>
<td>PRODUCER_THREAD</td>
<td>PRINT_TOOL_OBCS</td>
<td>10</td>
</tr>
<tr>
<td>6948</td>
<td>LOWER_BARRIER</td>
<td>PRODUCER_THREAD</td>
<td>PRINT_TOOL_OBCS</td>
<td>10</td>
</tr>
<tr>
<td>6948</td>
<td>HOLD_THREAD</td>
<td>PRODUCER_THREAD</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>7008</td>
<td>LEAVE_PROTECTED</td>
<td>PRINT_TOOL_THREAD</td>
<td>PRINT_TOOL_OBCS</td>
<td>10</td>
</tr>
<tr>
<td>7011</td>
<td>ENTER_PROTECTED</td>
<td>PRINT_TOOL_THREAD</td>
<td>STORE_OBCS</td>
<td>9</td>
</tr>
<tr>
<td>7047</td>
<td>LEAVE_PROTECTED</td>
<td>PRINT_TOOL_THREAD</td>
<td>STORE_OBCS</td>
<td>11</td>
</tr>
<tr>
<td>7051</td>
<td>ENTER_PROTECTED</td>
<td>PRINT_TOOL_THREAD</td>
<td>STORE_OBCS</td>
<td>9</td>
</tr>
<tr>
<td>7087</td>
<td>LEAVE_PROTECTED</td>
<td>PRINT_TOOL_THREAD</td>
<td>STORE_OBCS</td>
<td>11</td>
</tr>
<tr>
<td>7093</td>
<td>ENTER_PROTECTED</td>
<td>PRINT_TOOL_THREAD</td>
<td>PRINT_TOOL_OBCS</td>
<td>9</td>
</tr>
<tr>
<td>7109</td>
<td>SPORADIC_WAIT</td>
<td>PRINT_TOOL_THREAD</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7109</td>
<td>SELECT_AND_CONTEXT_S</td>
<td>PRODUCER_THREAD</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>7148</td>
<td>RESUME_THREAD</td>
<td>PRODUCER_THREAD</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>7187</td>
<td>CYCLIC_SUSPEND</td>
<td>PRODUCER_THREAD</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>7187</td>
<td>SELECT_AND_CONTEXT_S</td>
<td>CONSUMER_THREAD</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>7226</td>
<td>ENTER_PROTECTED</td>
<td>CONSUMER_THREAD</td>
<td>CONSUMER_OBCS</td>
<td>5</td>
</tr>
<tr>
<td>7251</td>
<td>LEAVE_PROTECTED</td>
<td>CONSUMER_THREAD</td>
<td>CONSUMER_OBCS</td>
<td>7</td>
</tr>
<tr>
<td>7251</td>
<td>ENTER_PROTECTED</td>
<td>CONSUMER_THREAD</td>
<td>BUFFER_OBCS</td>
<td>5</td>
</tr>
<tr>
<td>7281</td>
<td>LEAVE_PROTECTED</td>
<td>CONSUMER_THREAD</td>
<td>BUFFER_OBCS</td>
<td>8</td>
</tr>
<tr>
<td>7281</td>
<td>ENTER_PROTECTED</td>
<td>CONSUMER_THREAD</td>
<td>STORE_OBCS</td>
<td>5</td>
</tr>
<tr>
<td>7318</td>
<td>LEAVE_PROTECTED</td>
<td>CONSUMER_THREAD</td>
<td>STORE_OBCS</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 5.3 shows excerpts from the log of events generated by the scheduler simulator for the example system which are predicted to occur within the observation window between 6.8 and 7.4 millisecond.

The textual version of the simulation log reports on: the time of occurrence (TIME) of the logged event (expressed in μs); the type of event (EVENT); the running thread (THREAD) at the time of the event; the server involved in the operation (PO) where applicable; the active priority (PRIO) of the running thread at the time of the event, which therefore reflects the effect
of the priority ceiling emulation protocol on the active priority of threads entering and leaving servers.

The same log is graphically represented in figure 5.14, where solid blocks indicate the running status of a thread and grey areas are used to denote the occurrence of run-time system services in support of the application (e.g.: to enter / leave a server, to release a thread, etc.)

![Scheduler Simulator Output (Gantt)](image)

Figure 5.14: Scheduler Simulator Output (Gantt).

### 5.8 Summary

This chapter has illustrated the distinguishing features, performance and operation of our engineering approach to the development of new-generation real-time on-board software systems.

The structure of our presentation in this chapter has reflected the progression of the development process across the four stages B.1-B.4 described in section 4.3.1 in the construction of a simple example system. The demonstrative development has been assisted by use of the enabling technology developed by European Space Agency contract 9848/92/NL/FM under the guidance of the author of this thesis. ([Saab Ericsson Space, 1997] provides a comprehensive account of the result of that study contract.) The block diagram of the toolset developed as part of that contract is depicted in figure 4.6. The very existence of this toolset contributes to demonstrate the feasibility and practicality of our concept. Moreover, the direct utilisation of the toolset in the construction of the example system has allowed us to illustrate:

- the use of the HRT-HOOD design method [Burns and Wellings, 1995b] as the centre of our proposed development process and the support the method provides for the structured definition of the real-time commitments of the system as well as for the enforcement of the constructive prescriptions associated with the HRT-HOOD Computational Model;

- the support for the automated generation of a concurrent Ada program from an HRT-HOOD design in a fashion which preserves the desired real-time attributes of the system;

- the use of response time analysis as the source of early and continuous feedback to the consolidation of the design and implementation of the system.
By way of this presentation, we have demonstrated that our development process achieves the objective of this thesis as set out in section 1.4 and fulfils the requirements C.1-C.4 discussed in section 4.2.3. In particular, we have shown that our engineering approach:

1. supports the construction of real-time systems in accordance with the prescriptions of a well-defined Computational Model and ensures the compliance of the corresponding implementation, as demanded by requirement C.1; (the relevant activities were illustrated in section 5.3 and 5.4);

2. provides for an iterative and incremental process of verification of the system in the time-domain, as demanded by requirement C.2; (the relevant activities were illustrated in section 5.5, 5.6 and 5.7);

3. supports the analysis and interpretation of the effects of modifications, adaptations and enhancements to the real-time behaviour of the system, as demanded by requirement C.3; (the relevant activities were illustrated in section 5.7);

4. is implemented in an evolutionary fashion by means of simple enhancements to the HOOD and Ada 83 technology presently in use at European space industry, as demanded by requirement C.4.
Chapter 6

Proof of Concept

6.1 Historical Note

The early elements of the engineering approach we presented in chapter 4 and illustrated in chapter 5 trace their root back to [British Aerospace, 1993]. [British Aerospace, 1993] was a research project set out by the European Space Agency at the beginning of the 90s to devise ways to improve the productivity of HOOD [HTG, 1993] and Ada [ISO, 1987] technology in the construction of new-generation real-time on-board systems.

The author's involvement in that project and the concurrent rise of the "cheaper, faster, better" paradigm discussed earlier in this document effectively prompted the initial motivation to this thesis. On the other hand, the author's mandate at the Research & Technology Centre of the European Space Agency provided an excellent opportunity for the author to further his initial vision and bring it to the maturity documented by this publication.

In the following, we will briefly outline the strategy we have adopted to secure the establishment of our concept on solid foundations. Subsequently, we will present the experimental evidence which we have produced to demonstrate the performance of our overall engineering approach.

6.2 Concept Establishment

Since the early start associated with the involvement in the cited project at the beginning of the 90s, the development of our concept has progressed across three distinct temporal phases ("definition", "consolidation" and "maturity") respectively characterised by incremental levels of concern ("concept validation", "technology development" and "technology transfer").

Definition phase. The definition phase was entirely comprised within the scope of the study project documented in [British Aerospace, 1993]. The project (hence, the phase) included the
successful development of a pilot application (later referred to as the Olympus case study).

**Consolidation phase.** The seminal results of [British Aerospace, 1993] were subsequently consolidated by a number of other activities which built upon that initial core. The cornerstones of the "consolidation" phase were marked by: (a) the Eureca case study [Intecs Sistemi, 1994a, Intecs Sistemi, 1994b], at the level of concept consolidation; and (b) the development of an actual HRT-HOOD design support tool as well as of the ERC32 toolset, for the implementation of the related enabling technology. [Intecs Sistemi, 1996a] describes the operation of the former tool; [Saab Ericsson Space, 1997] and [Logica, 1997] report on the construction and operation of the latter. In the context of this document, the contents of the ERC32 toolset have been first conceptually introduced in chapter 4 and later utilised in the example presented in chapter 5.

**Maturity phase.** At various points of the development, the confidence increasingly gained in the concept and the success with the pilot applications permitted us to expose successive increments of the proposal to a wider dissemination. This was accomplished by way of presentations, demonstrations and technical publications. [Bailey et al., 1993], [Vardanega, 1994], [Vardanega, 1995], [Vardanega, 1996], [Vardanega, 1997] and [Vardanega, 1998] are some of the key publications in this respect. The important promotion effort deployed had distinct consequences. In fact, independently from the strategy we were deploying, an industrial project, ERA [Heemskerk and Schoonejans, 1997], autonomously decided to adopt the core of our proposed concept as the centre of their software development plan. Since its development was outside our development programme, the experience being made with that project can be considered the first true step into the "technology transfer" dimension.

Figure 6.1 depicts the overall chronology of this development.

![Figure 6.1: Chronology of Development.](image-url)
6.3 Approach to Assessment

Our assessment of the proposed concept has taken place from analysis of three distinct and complementary perspectives:

1. *process concept*: the suitability and performance of the proposed development process; this involved the scrutiny of each of the four stages of the proposed process: (i) real-time design, (ii) binding to Computational Model, (iii) static analysis and (iv) feedback to design;

2. *enabling technology*: the suitability and performance of the use of a restricted form of Ada tasking for the implementation of the chosen Computational Model; the tasking features allowed included in our Computational Model were discussed in section 4.5.1; the resulting profile has recently gained international recognition under the guise of the so-called Ravenscar profile outlined in [Baker and Vardenega, 1997];

3. *industrial applicability*: the applicability of the proposed development concept to the real industrial domain; in particular, emphasis was placed on the *scalability* of the engineering approach to the increasing size and complexity of new-generation systems as well as on the *transferrability* of the concept to the industrial developer.

Table 6.1 schematically summarises the "check list" which captures the above concerns.

<table>
<thead>
<tr>
<th>Area</th>
<th>Concern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Concept</td>
<td>Real-time design</td>
</tr>
<tr>
<td></td>
<td>Binding to Computational Model</td>
</tr>
<tr>
<td></td>
<td>Static analysis</td>
</tr>
<tr>
<td></td>
<td>Feedback to design</td>
</tr>
<tr>
<td>Enabling Technology</td>
<td>Ada tasking profile</td>
</tr>
<tr>
<td>Industrial Applicability</td>
<td>Scalability</td>
</tr>
<tr>
<td></td>
<td>Transferrability</td>
</tr>
</tbody>
</table>

In the remainder of this chapter, we demonstrate the validity of the proposed development concept by reviewing achievements and lessons learned derived from the progression shown in figure 6.1. The review concentrates on the analysis of the areas of concern captured by the check list in table 6.1.

We have selected the Olympus, Eureca and ERA projects as the body of evidence for our proof of concept. This choice was motivated by two major considerations: firstly, the relative position of the selected pilot cases within the chronology of our development naturally corresponds to a progressive evolution of the "profile of expectation" placed on each of the project. secondly, those projects target complementary elements of the space application in our domain, which offers us the most convenient opportunity to cover the whole spectrum of issues addressed by table 6.1.

For each pilot case under consideration, we: (i) indicate what elements of the proposed concept the project placed emphasis upon; (ii) present the architecture of the target system; (iii)
outline distinctive elements of the implementation experience; and, finally, (iv) review the main results and lessons learned from the case.

6.4 Definition Phase: The Olympus Case Study

6.4.1 Positioning of the Project

Olympus was launched in July 1988 as the world largest and most powerful civil three-axis-stabilised communications satellite. The Olympus case study addressed the re-development of a representative subset of the attitude and orbit control software system embedded on board the European Space Agency’s Olympus satellite. The re-development of its attitude and orbit control software (AOC) was selected as pilot application, it being a distinct mission-critical component of the system which also included particularly representative elements of typical on-board embedded real-time software.

The main goal of the project was to build confidence in: (a) our vision of real-time design as the ability to construct a physical model of the system around a well-defined instance of our Computational Model; (b) the feasibility and practicality of using response time analysis as the means to statically verify the timing behaviour of the system; and (c) the fitness of the anticipated revision of the Ada 95 tasking model as the basis of the chosen instance of Computational Model.

Table 6.2 summarises the “expectation profile” of the project.

<table>
<thead>
<tr>
<th>Area</th>
<th>Concern</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Concept</td>
<td>Real-time design</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>Binding to Computational Model</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Static analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feedback to design</td>
<td></td>
</tr>
<tr>
<td>Enabling Technology</td>
<td>Ada tasking profile</td>
<td></td>
</tr>
<tr>
<td>Industrial Applicability</td>
<td>Scalability</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transferrability</td>
<td></td>
</tr>
</tbody>
</table>

The project was carried out over the period November 1991 through February 1993 by a consortium involving industrial partners (British Aerospace and York Software Engineering, UK) and academia (the Real-Time Group at the University of York, UK) under funding and coordination from the Agency. The author of this thesis was actively involved in the technical coordination of the work. The complete account of the activity is reported in full as an annex to [British Aerospace, 1993]. [Bailey et al., 1993] and [Vardanega, 1994] provide a more concise account of the main results obtained from the activity.

The span of the project broadly covered the SR–ST segment of the V model and used the functional architecture of the original system as the basis for the re-development exercise.
The Olympus software system was re-implemented in an experimental language based on the Ada 83 standard augmented with enhancements foreseen by the Ada 9X language revision. The experimental language was supported by the York Ada Compilation System. Language enhancements and compiler features are described in [York Software Engineering, 1992].

![Diagram of the Olympus AOC Hardware Architecture](image)

Figure 6.2: The Olympus AOC Hardware Architecture.

### 6.4.2 System Architecture

The original Olympus AOC subsystem is controlled by the Control Electronics Unit (CEU). The core of the subsystem is represented by the Spacecraft Microcomputer Module (SMM) a processing unit based on the TI SPB 9989, a 16-bit microprocessor run at 4 MHz and equipped with 32 kilo-words of memory. The original flight software was written in SPB 9989 assembly. For the purposes of the case study, we replaced the SMM with two VME boards containing one 68020 processor run at 16.67 MHz with cache disabled, one 68881 floating point coprocessor, 1 megabyte RAM, timers and dedicated chips to support communications over the external data bus. Figure 6.2 displays the block diagram of the Olympus AOC subsystem. The box tagged 68020 in the figure represents the processing unit we used in the case study in the place of the original SMM.

The AOC subsystem acquires and maintains the position and orientation required for the satellite in the current mode of operation. Different modes of operation reflect different phases of the mission (e.g.: orbit, manoeuvre). The Olympus AOC supports six modes of operation: only the Normal Mode of the satellite was addressed in the scope of the case study as it was the most complex of them all and the most representative of the nominal operation of the satellite.

The Earth pointing accuracy required for the Normal Mode is maintained by a 200-millisecond periodic task that controls the current attitude and operates the four Reaction Wheels accordingly. Attitude data is received approximately every 100 milliseconds. Possible orbital drift is corrected by a periodic task activated twice per day for an interval of 15 minutes.
These regular activities are complemented by activities encoded as tasks occurring at less frequent or irregular intervals. Handling of traffic on the external serial bus is one of these activities: sporadic in nature, and activated with an interarrival time not inferior to 960 μs. The Telemetry & Telecommand Collector unit is another source of sporadic activities: it routinely requests status information from other AOC software components in order to keep ground informed of the current spacecraft status. The minimum period of issue of status information requests by the TTC unit is 62.5 milliseconds.

A dedicated service is provided for the handling of telecommands sent from ground to command the execution of specific on/off operations (e.g.: enable, disable, actuate) over selected devices. The arrival of such telecommands is sporadic in nature, with a minimum interarrival time of 190 milliseconds.

On the whole, the Olympus AOC software system exhibits a predominantly cyclic nature with a significant presence of interrupt-driven and generally sporadic activities. As a reflection of this, the scheduling model adopted for the original implementation was essentially cyclic. Its timeline was predominately occupied by synchronous tasks with slots reserved for the execution of asynchronous tasks. Limited use of preemption allowed the execution of interrupt-driven tasks in preference to synchronous and asynchronous tasks.

### 6.4.3 Implementation Experience

An early version of the HRT-HOOD method was used to steer the design phase of the pilot project. Owing to the lack of a real HRT-HOOD supportive tool, the method was applied by overimposing design conventions to a commercial HOOD tool. The resulting design experience mainly served to assess the expressive power of the proposed HRT-HOOD method with respect to the actual real-time needs of the target system. Table 6.3 relates the design break-down of the case-study system to the original system.

<table>
<thead>
<tr>
<th>Development Viewpoint</th>
<th>Original</th>
<th>Case Study</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design Aspects</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of design components</td>
<td>28</td>
<td>62</td>
</tr>
<tr>
<td>No. of terminal objects</td>
<td>-</td>
<td>43</td>
</tr>
<tr>
<td>No. of cyclic objects</td>
<td>26</td>
<td>9</td>
</tr>
<tr>
<td>No. of sporadic objects</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>No. of protected objects</td>
<td>-</td>
<td>14</td>
</tr>
<tr>
<td>No. of passive objects</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td><strong>Implementation Aspects</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of active tasks</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>No. of interrupt tasks</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>No. of passive tasks</td>
<td>-</td>
<td>17</td>
</tr>
<tr>
<td><strong>Total no. of tasks</strong></td>
<td>28</td>
<td>32</td>
</tr>
</tbody>
</table>

The fitness of the chosen Ada-based Computational Model for the target application domain was challenged by the requirement to handle interrupts off the external data bus with the worst-
case inter-arrival time of 960 μs. The initial HRT-HOOD design of the system had utilised one interrupt sporadic object to handle the interrupt off the bus, acquire the incoming data and trigger one software sporadic object charged with the dispatching of the interrupt data to the end users. The total service time needed by those two sporadic components to complete the required data transfer would take some 590 μs out of the 960 μs worst-case bus cycle. This was obviously unacceptable for it alone consumed over 60% of the available time slot, thus leaving insufficient room for the remainder of the system processing.

A large proportion of that overhead was due to the prototype nature of the modified Ada system being employed: the same model implemented on the recently completed ERC32 software development system [Saab Ericsson Space, 1997, Logica, 1997] and run on the more powerful ERC32 processor would now experience the more modest worst-case overhead in the range 170 to 390 μs. This notwithstanding, the problem denoted that the adopted mapping would be too heavy-weight as the handling of every interrupt off the bus would incur the fixed overhead of two extra context switches prior to the actual data dispatch.

Timing problems of this kind are not rare for on-board embedded systems and are often dealt with only from an execution performance point of view. Following our approach, we had the ability to analyse the problem from the design perspective also. This allowed us to devise one straightforward modification to the logical model of the system which would cleanly overcome the problem: the interrupt sporadic object would store the data acquired off the bus into a protected buffer object while one independent cyclic object would fetch the data at an appropriate rate and dispatch it to the end users. That simple modification resulted in a leaner design and a lighter-weight implementation.

### Table 6.4: Case Study vs Original Software (Performance Viewpoint).

<table>
<thead>
<tr>
<th>Performance Viewpoint</th>
<th>Original</th>
<th>Case Study</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Features</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor</td>
<td>SBP9989</td>
<td>MC68020</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>4 MHz</td>
<td>16.67 MHz</td>
</tr>
<tr>
<td>Throughput (MIPS)</td>
<td>0.075</td>
<td>2.5</td>
</tr>
<tr>
<td>Available RAM (bytes)</td>
<td>32K</td>
<td>1M</td>
</tr>
<tr>
<td>Implementation language</td>
<td>assembler</td>
<td>Ada83 plus ext.</td>
</tr>
<tr>
<td>Scheduling model</td>
<td>cyclic</td>
<td>preemptive priority based</td>
</tr>
<tr>
<td><strong>Memory Requirements (bytes)</strong></td>
<td></td>
<td>(unspecified)</td>
</tr>
<tr>
<td>Operating System</td>
<td>6,272</td>
<td>28,672</td>
</tr>
<tr>
<td>Application</td>
<td>13,714</td>
<td>60,940</td>
</tr>
<tr>
<td>Data Area</td>
<td>1,724</td>
<td>36,624</td>
</tr>
<tr>
<td>Total</td>
<td>21,710</td>
<td>126,236</td>
</tr>
<tr>
<td><strong>Processing Requirements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Predicted CPU load (WCET)</td>
<td>89.04%</td>
<td>83.16%</td>
</tr>
<tr>
<td>Typical CPU load</td>
<td>61.59%</td>
<td>37.74%</td>
</tr>
</tbody>
</table>

In February 1993, the correct operation of the case-study software system was successfully demonstrated on the Olympus Engineering Test Environment. Table 6.4 relates implementation aspects of the re-engineered system to equivalent aspects of the original system.
Upon completion of the main redevelopment exercise, the software was ported to a real space-qualified 16-bit processor in order to obtain a better qualitative appreciation of its actual feasibility over more representative space technology.

The porting exercise required some adjustments to the implementation of the binding of the application to the chosen Computational Model, as some of the needed language extensions were not available on the commercial Ada 83 cross compiler in use. This notwithstanding, the port succeeded and resulted in a crisp 45% reduction in the size of the software. Yet, that would still suggest – by extrapolation – that the entire Olympus AOC software system would possibly require about 160 kilo-bytes of memory and consequently demand an extended-memory 16-bit flight configuration.

Typical 16-bit flight configurations are normally equipped with no more than 128 kilo-bytes of RAM. This critical limit, in fact, presently represents the most common compromise between the high purchase cost of space-qualified memory, the containment of the corresponding power consumption profile and the technical complexity of explicitly programming allocation and access to added memory on the target processor (cf., for example, the experience reported in this respect by [Riehle, 1994]).

Table 6.5 summarises the main results from the port.

<table>
<thead>
<tr>
<th>System Features</th>
<th>Ported Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>MA31750</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>16 MHz</td>
</tr>
<tr>
<td>Throughput (MIPS)</td>
<td>2.4</td>
</tr>
<tr>
<td>Available RAM (bytes)</td>
<td>128 k</td>
</tr>
<tr>
<td>Run-time system support</td>
<td>standard Ada 83</td>
</tr>
</tbody>
</table>

**Implementation Aspects**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of active tasks</td>
<td>18</td>
</tr>
<tr>
<td>No. of interrupt tasks</td>
<td>1</td>
</tr>
<tr>
<td>No. of passive tasks</td>
<td>13</td>
</tr>
<tr>
<td>Total no. of tasks</td>
<td>32</td>
</tr>
</tbody>
</table>

**Memory Requirements (bytes)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>11,546</td>
</tr>
<tr>
<td>Application</td>
<td>49,220</td>
</tr>
<tr>
<td>Data Area</td>
<td>9,566</td>
</tr>
<tr>
<td>Total</td>
<td>70,332</td>
</tr>
</tbody>
</table>

### 6.4.4 Discussion

**Real-time design:** The pilot project confirmed that the execution behaviour of AOC systems like Olympus is predominantly cyclic and time-driven, with a tiny yet distinct component of sporadic event-driven activities. These requirements appear to be well captured by the notion of cyclic and sporadic object supported by the HRT-HOOD method.
The Olympus case also shows that on-board systems may include cooperating activities that share access to common resources and need support for data-oriented synchronisation.

The case study proved that the HRT-HOOD notion of protected object derived from the revised Ada 95 language standard [ISO, 1995] suits this need conveniently, while plain HOOD did not. In particular, table 6.3 shows that about one third of the terminal objects in the design of the Ada system were of protected type whereas none possessed any similar characteristics in the original design.

**Static analysis:** The use of HRT-HOOD as centre of our engineering approach facilitates the implementation of systems that can be analysed for their timing characteristics at all stages of development. The key to this ability is in the ease with which the descriptive model of the system subject of the chosen form of response time analysis may be derived from an HRT-HOOD design.

In order to exploit the potential, prototype tools for Worst-Case Execution Time (WCET) determination [Forsyth, 1992], response time analysis and scheduling simulation were designed and implemented as "concept demonstrators" and used to assist the re-development of the Olympus system.

The performance of the WCET extractor tool is particularly critical to the usefulness of timing analysis: an overly pessimistic WCET profile may in fact result in too conservative a design and therefore achieve poor utilisation of the available processing resources. The predictions produced by the cited WCET extractor prototype were found to be about 40% pessimistic (cf. table 6.4). This was regarded as an acceptable performance for a concept demonstrator. Further investigations determined that only a core component of the overall pessimism (in the order of 20%) actually originated from the tool itself, while the remainder was easily explained on detailed analysis of the application behaviour.

**Ada tasking profile:** The choice to base the HRT-HOOD Computational Model on the profile described earlier in chapter 4 has the distinct advantage of minimising the semantic distance between the design objects and the underlying implementation constructs.

- The Olympus experience shows that this property enables the designer to master the complexity and the implications of the modifications and amendments which the design may require on occurrence of feedback from analysis and testing.

- The Olympus experience also shows, however, that next to a number of apparent process-related benefits, the choice to base the HRT-HOOD Computational Model integrally around an Ada implementation may have non-negligible implementation costs on platforms subject to severe resource constraints.

- The port to the 16-bit space-qualified processor has denounced the important increase in the amount of memory required for the Ada version of the system.

- The design modification introduced to overcome the excess overhead incurred in the processing of interrupts off the data bus has confirmed the heavy-weight nature of old Ada 83 technology.
Both drawbacks may hinder the applicability of the concept to implementations targeting the old 16-bit space technology, which is unlikely to improve much further. However, we expect both concerns to be dismissed by the advent of the new 32-bit technology.

**Productivity considerations:** The size of the application software component developed as part of the project exceeded 3,600 Ada statements, for an effort of about 2,100 hours deployed across a sizeable proportion of the SR–ST segment of the V model. Little can be concluded from these figures, however, as the development team in the case study had prior involvement in the real Olympus project (which lessened the complexity of the SR and AD phases), while the implementation progressed virtually in parallel with the 'debugging' of the HRT-HOOD design method (which affected the productivity of the DD phase).

### 6.5 Consolidation Phase: The Eureca Case Study

#### 6.5.1 Positioning of the Project

The Eureca case study aimed at complementing the contributions made by the Olympus case study in several important respects.

It addressed the re-development of the data handling control system of the European Space Agency's Eureca orbiting platform, while the Olympus case study had addressed the attitude and orbit control part of the on-board system. This was intended to expose the proposed concept of real-time design to the entire spectrum of the typical on-board service functions: as the Olympus project had covered the time-driven component of the typical on-board system, the Eureka project would now cover the event-driven component.

Second, the project marked a distinct step forward to the consolidation of the development strategy through exposure to the scrutiny and application by a new engineering team with no prior experience with the method. In particular, special attention in the project was devoted to: (a) determine the most effective binding path from the HRT-HOOD design to the chosen Computational Model; and (b) assess the benefit of using the response from static analysis to guide the occurrence of design and implementation iterations.

The project also placed emphasis on the capture and consolidation of the requirements on the needed enabling technology. This paved the way to the successive phase of technology development in which we accomplished the implementation of a full HRT-HOOD-supportive design tool [Intecs Sistemi, 1996a] as well as of the ERC32 toolset presented in [Vardanega, 1996] and [Logica, 1997].

Table 6.6 summarises the "expectation profile" of the project.

The project was carried out from January 1993 until April 1994 by Intecs Sistemi (Italy) under funding and coordination by the European Space Agency. Also in this case, the author of this thesis played a key role in the conception and technical coordination of the work. The complete account of the activity is reported in full in [Intecs Sistemi, 1994a] and [Intecs Sistemi, 1994b], whereas a more succinct summary is presented in [Vardanega, 1995].
### Table 6.6: Expectation Profile (Eureka Case).

<table>
<thead>
<tr>
<th>Area</th>
<th>Concern</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Concept</td>
<td>Real-time design</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Binding to Computational Model</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Static analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feedback to design</td>
<td>✓</td>
</tr>
<tr>
<td>Enabling Technology</td>
<td>Ada tasking profile</td>
<td></td>
</tr>
<tr>
<td>Industrial Applicability</td>
<td>Scalability</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transferrability</td>
<td></td>
</tr>
</tbody>
</table>

### 6.5.2 System Architecture

The original DHC software was written in C and was comprised of 22 concurrent tasks run under a customised version of a commercial real-time executive implementing preemptive priority based scheduling.

In addition to performing command and control of the platform and payload equipments, the Eureka DHC maintains the telecommand / telemetry (TC/TM) data exchange with ground.

The hardware architecture of the DHC is shown in figure 6.3. The system includes two processing units in cold redundancy which interface a mass memory unit (denoted as PPU and MBM, respectively, in the figure). The redundancy management, which is entirely hardware-based, is performed by the Master Reconfiguration Unit (MRM). Various user terminal equipments are connected to the PPU via a serial bus (DHS Bus in the figure).

The PPU is based on an 80C86 processor run at 5 MHz and equipped with 128 kilo-bytes of RAM and 16 kilo-bytes of ROM. In addition to that, the MBM unit provides 16 mega-bytes of total capacity physically organised as a cluster of 32 packages of 512 kilo-bytes each. The first of those packages contains a self-standing copy of the ROM-stored image of the flight program to support emergency bootstrap and post-mortem analysis. All the remainder memory in the MBM is used as temporary storage for large blocks of data and context information produced by the scientific experiment packages installed on board and periodically transmitted to ground for transformation and analysis.

The MBM unit is connected to the PPU through seven high-speed serial interfaces. Five of those are placed under the direct control of the on-board software while two are directly operated in DMA by dedicated hardware peripherals.

The Eureka DHC operates two TC links at 2 kilo-bit per second, two low-speed TM links also at 2 kilo-bit per second and two high-speed TM links at 256 kilo-bit per second. The high-speed links are operated in hardware during the periodic dump to ground of context and experiment data. The maximum overall sustained rate of the software-processed TM is limited to 42 kilo-bit per second.

The overall TC / TM data traffic to and from the PPU flow across two 4-channel DMA controllers and three interrupt controllers which are to multiplex over 20 interrupt lines coming from several I/O peripherals. During operation, interrupts would arrive in burst with worst-case interar-
rival time ranging between 8 and 96 milliseconds. As a result, the typical profile of the processing activities on board the Eureka DHC is distinctly I/O intensive and predominantly interrupt-driven.

The most critical interrupt-triggered sporadic activities to be performed by the system includes the following: (a) the management of incoming communications from the DHS bus, the arrival of which is signalled by a hardware interrupt with an average interarrival time of 16 milliseconds and a minimum of 8 milliseconds; and (b) the management of the End-of-Transfer signal generated off the bus upon completion of synchronous DMA transfers, which entails the verification by the PPU software of the correct completion of the data transfer.

The other main activities have the following timing requirements: (c) the Packet Content Manager, which collects the TM packets produced by on-board users and must, in the worst case, be able to assemble and dispatch one 512-byte telemetry frame at every 96 milliseconds; (d) the Telecommand Manager, which must be able to support the maximum uplink rate of one 256-byte TC packet per second; (e) the Master Schedule Manager, which must ensure the timely activation of time-tagged mission control commands and initiate at most one such command per second; (f) the Monitoring Service Manager, which must complete three bus transfer operations over a regular period of 256 milliseconds.

The rest of the software system is made up of a number of other activities with less critical timing requirements and non-marginal processing demands (e.g.: Command Interpreter, Command Executor, Checkpointing, etc).

Overall, the timing and processing requirements of the entire system is such that the peak load would typically occur within the first 96 milliseconds of every 1.056-second period of execution (equivalent to the major frame of the nominal system operation). This consideration was
retained in the critical-instant condition established in the adopted model of scheduling analysis. A graphical representation of the principal scheduling events that would occur within the critical 96-millisecond interval is shown in figure 6.4.

Figure 6.4: Worst-Case Software Schedule.

6.5.3 Implementation Experience

The scope of the case study was limited to slightly less than 70% of the original software system. However, the functions excluded from the implementation were not time-critical, hence not particularly relevant for the purpose of the exercise.

The HRT-HOOD re-design of the system was made up of 35 terminal objects. The HRT-specific component of the system included 4 protected, 4 cyclic and 6 sporadic objects. The design also included 11 non-critical active and 10 passive conventional HOOD terminal objects. The corresponding Ada 83 implementation was made up of a total of 35 tasks with the following break-down: 10 critical tasks (for as many cyclic and sporadic terminal objects); 14 passive tasks (for 4 protected terminal objects plus one object control structure for every cyclic and sporadic object in the system); and 11 non-critical tasks (for as many non-critical active terminal objects).

The overall development effort was split in two consecutive phases. The first phase encompassed the SR-AD segment of the V model and included a first iteration of the DD/UT phase performed against a non-representative platform but equipped with a more advanced cross-development environment. In this set-up, all of the external interfaces required for the operation on the actual target environment had to be simulated in software. The second and final phase covered the whole IT-ST segment of the V model and included the port of the software to the actual engineering model of the spacecraft.

The transition between the first and the second phase was accomplished on successful completion of unit testing and verification of the feasibility of the timing behaviour expected on the final target. Figure 6.5 displays a sample of the results from timing analysis performed on the software at the transition between the two phases.

The results shown in figure 6.5 indicate that the system is fully schedulable on the development platform (i386) while it is not on the final target (i86) where the the predicted processing load
would top the capacity of the processor and several low-criticality activities might also miss their deadline. (The performance ratio between the two platforms was roughly estimated to a factor of 8.)

A few design modifications and code optimisations were thus necessary prior to proceeding with the port to the actual target environment. The required corrective measures amounted to a further critical review of design concepts and code structures in conjunction with detailed inspection of the possible causes of unschedulability indicated by the analysis tools. The fixes were successful and allowed the porting to complete as planned.

The corrective measures were complemented by extensive optimisation of the data processing algorithms and the data structures employed for the final version of the system. This action bought the system a further reduction of about 15% in code size and of some 25% in the incurred runtime overhead, thereby achieving better responsiveness to external events so crucial to a system as interrupt-intensive as Eureca.

Table 6.7 displays a few comparative values determined from the original system and its redevelopment; the latter were taken from both the interim (i386-based) and the final standpoint. These figures should be compared with caution, as they obviously reflect the important differences between the respective developments. For example, the Ada figures refer to about 70% of the original system, while the C figures refer to its entirety. Similarly, the data area required for the C version is slimmer than that of the Ada system, as the former could offload some of its data structures to the mass memory unit, while that was not possible for the latter.

Table 6.7 shows that the major drawback with the Ada version was definitely placed on the significant inflation of required memory: considered the relative proportion of the two developments, the size of the Ada version almost doubled the size of the C version. Retrospectively, this

---

**Figure 6.5: Scheduling Analysis Reports.**
Table 6.7: Original Software vs Redevelopment.

<table>
<thead>
<tr>
<th>System Features</th>
<th>Original</th>
<th>Interim</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>80C86</td>
<td>80(3)86</td>
<td>80C86</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>5 MHz</td>
<td>16 MHz</td>
<td>5 MHz</td>
</tr>
<tr>
<td>Available RAM (bytes)</td>
<td>128K</td>
<td>1M</td>
<td>128K</td>
</tr>
<tr>
<td>Implementation language</td>
<td>C</td>
<td>Ada 83</td>
<td>Ada 83</td>
</tr>
<tr>
<td>Scheduling Model</td>
<td>preemptive priority based</td>
<td>preemptive priority based</td>
<td>preemptive priority based</td>
</tr>
<tr>
<td>Memory Requirements (bytes)</td>
<td>100%</td>
<td>70% (unopt)</td>
<td>70% (opt)</td>
</tr>
<tr>
<td>Operating System</td>
<td>6,094</td>
<td>22,246</td>
<td>20,148</td>
</tr>
<tr>
<td>Application</td>
<td>53,372</td>
<td>79,168</td>
<td>67,908</td>
</tr>
<tr>
<td>Data Area</td>
<td>20,712</td>
<td>49,032</td>
<td>39,926</td>
</tr>
<tr>
<td>Total</td>
<td>80,178</td>
<td>150,446</td>
<td>127,982</td>
</tr>
<tr>
<td>Processing Requirements</td>
<td>90.6%</td>
<td>14.1%</td>
<td>69.8%</td>
</tr>
</tbody>
</table>

should be attributed to three distinct factors:

1. the Ada version had to retain all data structures in RAM, while these were offloaded to the MBM unit in the C version;

2. the use of tasking on embedded Ada 83 technology tends to require a larger memory space for the run-time system than typically needed by custom executives (as used for the C version);

3. the HRT-HOOD method apparently induces a more structured design of the system than achieved with conventional HOOD; in the case of the Eureca project (similarly to the Olympus case) this resulted in a system comprised of a greater number of active terminal objects with a predominant HRT connotation: the method prescribes that these objects be mapped to task-based Ada units and this obviously increases the ultimate size of the application.

The size of the Ada version could most certainly be further reduced, as memory optimisation was not the main driver to the overall development exercise. However, similarly to what was noted for the Olympus experience, the concern remains that the Ada implementation of the entire control system on the present 16-bit space-qualified technology would possibly demand a RAM size in excess of the critical limit of 128 kilo-bytes.

Wherever possible, aspects of execution performance were also considered. This effort was somewhat hindered by the limited details available in this respect on the C system. Still, it appeared that, for the most part of the on-board services performed by the DHC system, the C version was consistently faster than the optimised Ada version (about 35% faster on average). This was not surprising, as such operations were mostly related to the management of hardware-near data structures, where Ada may be significantly slower than C. This deficiency, however, did not affect the overall adequacy of the operation of the Ada version (which shows one more time that raw performance is not a necessary ingredient of real-time).
6.5.4 Discussion

Real-time design: The aim of the case study was to assess the effectiveness of the proposed approach to the SR-DD segment in the descending branch of the V model. In particular, the adopted process differed from the conventional development model in two distinguishing aspects: (1) the use of real-time analysis since the early stages of development as the means to permit the joint consolidation of the logical and physical model of the system advocated by the design framework concept described in section 4.2.1; and (2) the consequent ability to defer the transfer to the physical target until after preliminary verification of the correct functional and real-time behaviour of the system, thereby reducing the costly extent of development performed on the physical target. Both enhancements proved clearly beneficial.

Binding to Computational Model: The use of the semantic contents of the chosen Computational Model facilitates the automated generation of the concurrent structure of the application program directly from the HRT-HOOD design. This was expected to increase the productivity of the DD–IT segment and consequently that of the entire process, and this was effectively the case.

Feedback to design: The Eureca case confirmed that the generation of increasing knowledge about the system's operation and performance occurring along with the progress of the development is an important agent of design and implementation iterations. This phenomenon is captured by the process flows shown in figure 4.1 and structured in the process concept depicted in figure 4.2. This revealed an important asset of the chosen approach, for it prevented the uncontrolled occurrence of development iterations which may be greatly disruptive of the consistency of the work already performed.

Productivity considerations: The Eureca case study was carried out in two sequential phases which spanned the total duration of 16 months. The initial development phase, which included design, implementation and host-based integration, took about 12 months. The final phase, which included the port of the software to the physical target and the execution of flight-worthy acceptance tests, was accomplished in 4 months. The effort absorbed by the portion of development performed on the physical target (broadly corresponding to the IT–ST segment of the V model) amounted to about 25% of the overall total, hence significantly less than normally experienced (cf. the typical effort distribution for that segment as shown in figure 3.1, which ranges about 55-60%). The comparatively short duration and the smooth progress of the ascending branch of the project reflected the performance of the proposed development strategy and confirmed the postulates introduced earlier.

In fact, the productivity boost observed in the case study was not limited to the DD–IT segment but reverberated across the entire span of the project, which covered the SR–ST segment of the V model. The overall activity took 16 months to deliver a total amount of little over 12,000 source lines with an average team of 1.25 persons. The mean productivity therefore exceeded 4 source lines per hour. That compared well with the productivity rate typically observed for the delivery
of current-generation flight systems, which, for systems comprised of 15-25,000 source lines, lies in the range 0.75-1.0 source lines per hour.

These figures as definitely meaningful, as the team had no prior involvement in the real Eureca project nor any prior experience with the proposed engineering approach.

The performance achieved in the project may thus rightly be regarded as very encouraging even though the two figures cannot be directly compared, as the latter reflect the exposure to far more demanding environmental conditions.

6.6 Maturity Phase: The ERA Project

6.6.1 Positioning of the Project

The success with the case studies discussed in section 6.4 and 6.5 permitted a wider dissemination of the proposed development approach. We have already mentioned in section 6.2 that technical publications such as [British Aerospace, 1993], [Vardanega, 1994], [Vardanega, 1995], [Vardanega, 1996], [Vardanega, 1997] and [Vardanega, 1998] had a major role in this respect.

This important promotion effort prompted segments of the European space industry to consider the essential elements of the approach advocated by this thesis for the development of their new-generation on-board real-time systems.

Notably, the HRT-HOOD design method and the associated development strategy were adopted by an industrial consortium led by Fokker Space (Netherlands) for the development of the on-board control software system of the European Robotic Arm (ERA).

ERA is a nearly-symmetrical 11.3-meter-long manipulator arm which shall be operated, in orbit, in the construction and servicing of the Russian segment of the International Space Station Alpha. [Heemskerk and Schoonejans, 1997] provides an outline of the project objectives and the associated system concept.

The development project is funded by the European Space Agency and executed in cooperation with the Russian space organisation.

The ERA project has a completely foreign status to the strategic framework outlined in section 6.2. Yet, the autonomous decision made by its engineering authority has placed the project in the position to provide a crucial contribution to the determination of the actual maturity of our proposal.

In particular, the contribution we expect from the experience with the ERA project regards the assessment of: (a) the scalability of the proposed development approach to the functional and operational complexity of full-scale new-generation space systems; and (b) the effectiveness of the transfer of the engineering concepts and technology to a fully autonomous industrial context. Table 6.8 summarises our expectation profile in the respect of the project.

6.6.2 System Architecture

The arm consists of two identical wrist/end-effector units and an elbow/limbs unit. The ERA control computer (ECC) is embedded in the heart of the arm, located within the elbow/limbs
Table 6.8: Expectation Profile (ERA Case).

<table>
<thead>
<tr>
<th>Area</th>
<th>Concern</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Concept</td>
<td>Real-time design</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Binding to Computational Model</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Static analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feedback to design</td>
<td></td>
</tr>
<tr>
<td>Enabling Technology</td>
<td>Ada tasking profile</td>
<td></td>
</tr>
<tr>
<td>Industrial Applicability</td>
<td>Scalability</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Transferrability</td>
<td>✓</td>
</tr>
</tbody>
</table>

unit. The ECC is based on the ERC32 computing core. (The interested reader is referred to [ESA, 1992a], [Gaisler, 1994] and [Saab Ericsson Space, 1997] for more details on the objectives, features and operation of processor boards based on the ERC32 computing core.)

The ECC is an on-board embedded real-time system in many respects alike the systems discussed earlier in this thesis. The ECC, though, has the additional distinguishing property of being far more safety-critical than any conventional satellite system because of the vicinity of the arm to the inhabited modules of the space station.

The block diagram of the ERA control system architecture is shown in figure 6.6.

![Figure 6.6: The ERA Control System Architecture.](image)

The diagram shows that the ECC communicates with the subsystem components of the robotic arm across a dedicated serial bus (denoted as "internal bus" in figure 6.6). This bus has exactly the same function as the internal busses shown in figure 6.2 and 6.3 for the systems in the case studies.

A separate bus ("external bus" in the figure) is used for the telecommand and telemetry ex-
change between ERA and the space station. The external bus also channels the communication between the ECC and the arm-operation interface units. The arm can in fact be commanded from a control-post computer located within the Russian segment of the space station as well as from an external interface accessible to astronauts in extra-vehicular activity. Commands from either source reach the ECC through the external communication bus.

6.6.3 Highlights of Early Implementation Experience

The ERA software system is an on-board real-time embedded control system comprised of over 20 concurrent time-critical cyclic and sporadic tasks in addition to some 20 protected objects, globally organised in accordance with the prescriptions of our preferred Computational Model.

The system is designed using the HRT-GOOD method and implemented using the extended Ada 83 technology developed as part of [Saab Ericsson Space, 1997]. The ERA control application directly relies upon the Ada run-time system without the need for any foreign executable.

On the whole, the ERA on-board software system is projected to amount to about 55,000 source Ada lines, hence significantly greater than the typical size of on-board systems of current generation. The maximum sustained CPU load of the system is presently predicted to be inferior to 70%.

In full accord with the process concept depicted in figure 4.1 and 4.2, the ERA software engineering strategy adopts an explicitly iterative and incremental development approach.

The approach progresses across two major steps: (a) the initial physical model of the system – denominated "Version 0" in the ERA jargon – is first sketched out and verified by analysis of the anticipated timing requirements; (b) the model is subsequently enhanced by successive increments of design and implementation details and accompanied by repeated stages of timing analysis based on increasingly accurate knowledge about the system's behaviour.

The ECC development was at some point shattered by the malfunction of the initial hardware-software baseline. The known deficiencies of that technology were so important that they required the immediate redirection of the project to some alternate space-qualified technology. The ERC32 hardware and software technology were at that point selected as the definitive target of this redirection. The transition to the ERC32 technology occurred amidst the definition of "Version 0" of the system. The project has survived this major crisis and unanimously attributed merit of the smooth and successful technology switch to the added value provided by the chosen engineering approach. (This is publicly acknowledged for example by [Heemskerk and Schoonejans, 1997].)

The ERA development has recently successfully completed the first development iteration, out of the three currently planned over the total project duration of 30 months. The first iteration is to establish the complete software architecture of the system (in PSS-05 terms, the physical model) and verify its real-time feasibility by static analysis. The project is presently in the process of entering the second iteration, which is to implement the code in the object bodies of the physical model and ensure that their timing requirements fit the margins determined by the prior analysis. The ERA project has also made provisions for a fourth development iteration to be accomplished either as a pre-launch software update or as the in-flight addition (by differential upload) of the required extra functionalities to the system.
6.6.4 Discussion

Scalability: The properties and requirements of the ERA control system as above outlined and the associated development logic show that the proposed engineering approach can scale up from the confines of the systems discussed in section 6.4 and 6.5 to target systems characterised by greater size, criticality and functional complexity.

Transferrability: Likewise the most part of the projects funded by the European Space Agency, the ERA software development is a cooperative effort contributed by a number of companies on the basis of an agreed partitioning and allocation of the work. This particular aspect of the ERA experience is instrumental to highlighting one further distinctive property of the evolutionary approach we advocate in [Vardanega, 1997] and [Vardanega, 1998]. The proposed development strategy, in fact, combines: (1) the benefit of the hierarchical approach to system design promoted by the base HOOD method, which caters for the controlled partitioning of the development, with (2) the means to support centralised verification of the real-time properties of the system and their distributed enforcement at all the levels of development. Both properties constitute a crucial asset in the presence of cooperative work programmes as these tend to evolve in an inherently incremental and distributed fashion and consequently need the means to centrally ensure the coherency of the system's operation and the correctness of its real-time behaviour. The proposed development approach centred around the use of the HRT-HOOD design method supports both aspects efficiently.

In the case of the ECC, the on-board system has been functionally split in three hierarchical layers, the development of which has been allocated to different industrial teams. The three layers are as follows: (a) the "application layer", which interfaces the ECC to the command centres in the space station and handles commands, mission plans and contingency events; (b) the "service layer", which embodies the main control loops of the robot arm; and (c) the "operating system layer", which builds on the Ada run-time system and supports the communications between the ECC and the connected arm subsystems across the internal serial bus. The team developing the "service layer" is also charged with the centralised process of real-time verification of the entire system.

The smooth progress of the software development observed so far across the various partnership levels of the ERA project apparently confirms the added value and performance of the strategy.

The ERA project is presently in progress and is expected to complete by the end of 1999.

6.7 Summary

This chapter has presented experimental evidence in support of the feasibility and performance of the proposed development approach.

The evidence supplied to that effect was evinced from two distinct sources readily available to the author of this thesis: (1) a pair of case studies which were conceived and conducted in the frame of the research programme from which this proposal has originated; and (2) an on-going
real industrial project which is to implement operational on-board components of the international space station Alpha. The former, which were actively participated by the author of this thesis, were instrumental to the conception and consolidation of key elements of the proposal. The latter is proving especially relevant as the engineering authority in the project did autonomously decide to adopt the software development approach advocated by this thesis. That choice makes the ERA project the first real industrial activity reported to actively pursue the proposed approach.

The differing scope and goals of the projects allowed the lessons learned from their scrutiny to be nicely complementary. That broadened the spectrum and coverage of the resulting insight, which are briefly summarised in the following.

The case studies discussed in section 6.4 and 6.5:

- proved the correctness of the characterisation of typical on-board systems which was presented in chapter 2;
- showed that the instance of Computational Model introduced in section 3.4.5 and selected in section 4.3.1 as integral part of the proposed development approach suits well the type of applications embedded on board space systems;
- confirmed the claim made in section 4.2.1 that the development of embedded on-board real-time systems possesses a strong iterative connotation in contrast with the "cleansed" sequential view put forward by the applicable PSS-05 standard;
- indicated the usefulness of unifying the development of the logical and the physical model of the system within the single design framework supported by the HRT-HOOD method, in line with the claim in section 4.6;
- demonstrated that the proposed approach increases the ability of the descending branch of the PSS-05 development model to deliver a system which requires less onerous stages of testing in the ascending branch.

On top of that, the experience reported from the industrial activity presented in section 6.6:

- displayed the appeal and practicality of the evolutionary approach advocated by section 3.4.1 in the presence of important components of HOOD and Ada legacy;
- showed that the proposed development strategy can scale up to the size, functional and real-time complexity of new-generation systems;
- confirmed the productivity gain which may be obtained from the iterative and incremental nature of the proposed development method, in line with the claims anticipated in section 4.2.1.
Chapter 7

Concluding Remarks

7.1 Recalling the Thesis Objective

This thesis addressed the development of the real-time software embedded onboard new-generation satellite systems and the challenge presented on this by the rise of the "cheaper, faster, better" project paradigm.

Currently, the software development scenario of new-generation systems is dominated by the demand to:

1. allow for the reduction of the satellite development schedule from the present 3-4 years to 18-24 months [ESA, 1995b, Van der Ha, 1995];

2. achieve the on-board capability to deliver increased autonomy, responsiveness and mission product [Teston et al., 1997];

3. support the trend to increasingly more software-intensive systems, the size of which is predicted to rapidly rise from the current 15-25,000 source lines to the 50-60,000 of new missions under the same language baseline [ESA, 1996, Vega, 1993]

As a result of these demands, the productivity of the process shall have to at least quadruple, on account of the demand for twice as big software to be delivered in half the time. Concurrently, the operation requirements placed on the on-board system will increase the functional complexity and the real-time criticality of the software product. Moreover, the customer expectation is that all that be achieved within no less than the present envelope of dependability.

These requirements challenge the current software practice in two distinct ways:

- The verification effort deployed in the current software development process is dominated by dynamic testing typically performed in a sequential fashion on costly (hence, centralised) test infrastructures. This arrangement can hardly sustain the demand for increas-
ingly complex and time-critical systems to be delivered within a dramatically compressed schedule, other than at the cost of higher risks and lesser quality.

- The software architecture of conventional on-board systems is typically centred around rigid and inflexible cyclic scheduling. The monolithic nature of these architectures makes them ill-equipped to accommodate incremental builds, late changes and development iterations, which are the anticipated characteristics of the future scenario.

Our motivation was to remedy these specific problems by the definition of process and product enhancements to accommodate the increasing real-time complexity of new-generation systems in the changing development scenario. In doing so, we placed special emphasis on the industrial applicability and cost-effectiveness of the proposed solution.

In particular, our thesis objective was to devise and support an innovative engineering approach to the development of new-generation on-board embedded real-time systems, centred around:

1. a verification process geared to static real-time analysis, as a substitute for dynamic real-time testing, in conjunction with

2. an architectural paradigm based upon fixed priority preemptive scheduling, as an evolutionary amendment to the current practice which can attain the combination of flexibility and predictability required of new-generation on-board systems.

7.2 Reviewing the Accomplishments

We achieved the thesis objective in a step-wise fashion.

The first step towards our goal was achieved by introducing two distinct enhancements to the current software process, namely:

- the notion of Computational Model, as the means to augment the expressive power of the design definition phase to capture: (a) the real-time attributes and execution characteristics of the components from which the software system is to be constructed; (b) the real-time execution model which underpins the design of the system; and (c) the means of communication and synchronisation in place among components and between them and the external environment;

- the support for a unified design framework, embracing the whole spectrum of the real-time design activities (i.e. requirements analysis; design definition; design analysis and implementation) into a single, unfractured process stage within which the development can proceed in an iterative and incremental fashion and ordinarily respond to the feedback arising from the progressive consolidation of the system.

The subsequent step was accomplished by defining precisely the flow of activities that are to occur within the design framework and ensuring that they can be implemented in an evolutionary
fashion with respect to the current practice. Accordingly, we established that the segment of the process covered by the design framework is to progress around the following four stages:

1. **real-time design**: this stage of the process is initially entered with the definition of the problem as determined by the system specification (i.e. the user requirements); this stage encompasses the definition of the logical and the physical model of the system and focuses on the characterisation of the real-time requirements placed on the individual components of the system;

2. **binding to Computational Model**: this stage of the process is entered at all times the abstract design of the system is submitted to implementation; at this point, the application program corresponding to the current level of design is automatically extracted from the design tool and fed to the designated compilation system; the compilation system binds the program to the selected Computational Model (thereby ensuring the use of the appropriate programming interface) and determines the processing requirements associated with the worst-case execution time profile of the application components;

3. **static analysis**: this stage of the process is entered to statically determine whether the current implementation of the system is capable of meeting the real-time commitments of the corresponding specification when executed on the designated target platform; at this level, we perform static response time analysis upon a stylised representation of the system and associated real-time attributes as derived from the current design, and the worst-case execution time profile of the relevant program as generated by the compilation system;

4. **feedback to design**: this stage of the process is entered to review the results obtained from the earlier stage of static analysis and determine the corrective measures (if any) required to ensure that the system implementation meets the designated real-time requirements; the determinations established at this level are then fed back to the design level in order for the designer to keep current the actual real-time attributes of the system and either exit the design process or perform further steps of design (and / or implementation) increment and corrective iteration.

The segment of the process covered by our proposal fits well in the software process model presently in place at European space industry, for it operates upon the same initial input as the latter and delivers the expected output.

The third step of our thesis work was completed by the definition of the technology elements required to enable our process and by the implementation of these as evolutionary enhancements to the currently technology baseline at European space industry. This baseline is presently centred around the HOOD method [HTG, 1993] for the design and Ada 83 [ISO, 1987] for the implementation. We, consequently, designed, built and demonstrated a toolset comprised of the following components:

- an augmented version of the HOOD design method, called HRT-HOOD and defined in [Burns and Wellings, 1995b], directly supportive of our Computational Model, based on
fixed priority preemptive scheduling and the deadline monotonic scheduling theory estab-
lished by [Audsley et al., 1991] and [Audsley et al., 1993];

- a standard Ada 83 [ISO, 1987] cross compilation system augmented with Ada 95 real-
time features [ISO, 1995] and provided with: (i) a precise characterisation of the timing
behaviour of the run-time support to the Computational Model, and (ii) built-in capabili-
ties for the extraction of worst-case execution time profiles of the application components
[Thomson Software Products, 1995, Thomson Software Products, 1996];

- a static analysis tool [Spacebel Informatique, 1996, Logica, 1996] designed in accordance
with [Joseph and Pandia, 1986], [Audsley et al., 1991] and [Tindell et al., 1994] to perform
response time analysis of systems generated with the aid of the toolset; our process model
facilitates the iterative execution of this analysis stage on successive increments of the sys-
tem; the information returned from the analysis device feeds the confirmation of the desired
timing behaviour of the current system generation or, else, the determination of the neces-
sary corrective measures.

The fourth and final step of the thesis work was to demonstrate the performance of our en-
gineering approach. We did so by presentation of the experimental evidence gathered from case
studies which accompanied the consolidation of our proposal. In addition to that, we also re-
viewed the success experienced by an industrial project which recently decided to adopt our
development strategy.

At the conclusion of our work, we contend that this thesis has presented original contributions
to the real-time community in general and European space industry in particular in two distinct
respects:

1. it has assembled a number of state-of-the-art research results into a coherent, practicable
   and industrial-quality engineering strategy which arguably meets the emerging require-
   ments;

2. it has designed and implemented the necessary enabling technology and provided for its
   immediate availability and durability to the industrial community.

7.3 Future Perspective

We conclude this presentation by briefly reviewing the prospective of our proposal in the respect
of: (i) the relevance it bears to application domains other than space; and (ii) the work required to
further extend its coverage of the future development cycle of on-board systems in Europe.

7.3.1 Relevance to Other Application Domains

On-board embedded applications belong in the general category of embedded real-time systems.
The former represent a tiny proportion of the European software market (1% of the total volume
of the software business in Europe in 1994 according to [ESA, 1996]). The dimension of the latter, however, is steadily growing in a variety of application domains. These domains presently include avionics, telecommunications, process control, railway, automobile (where, for instance, already some 12% and rising of a conventional car is reportedly made of software-controlled electronics) while analysts project an even bigger growth in a variety of other applications (cf., for example, [Randell et al., 1994]).

All those application domains have a distinct real-time component. Furthermore, they are all governed by stringent commercial drivers and, consequently, challenged by the same "cheaper, faster, better" paradigm which has motivated this work. By the same token, they should equally be faced with the demand to minimise the hardware and harness required and with the consequent escalation of the software complexity. In principle, therefore, the issues discussed in this thesis and the proposed development strategy should indeed apply to these application domains, too.

Clearly, however, we must differentiate the applicability of our process model from that of the enabling technology we developed for it. We have no doubt that there is nothing preventing the applicability of the former. The applicability of the latter to domains other than the European space industry, instead, is far more difficult to determine, for it depends on the nature of the technology legacy in the specific domain.

The option for the evolutionary approach made us build our technology upon the HRT-HOOD method [Burns and Wellings, 1995b, Burns and Wellings, 1994] for the design and the Ada 95 language [ISO, 1995] for the implementation. Considering the complex blend of factors which determine the choice of technology for critical applications, it is difficult to conclude anything about the applicability of HRT-HOOD outside the confines of the HOOD community. Conversely, the language-related side of the issue is less controversial, partly because of the sufficient penetration of Ada in the real-time embedded market and partly as the integrity of our engineering approach depends on the assurance of the binding step which correlates the level of real-time design to the Computational Model of choice much more crucially than on the implementation language. (Yet, we showed that Ada fits very well in our process and that the Ada implementation of our Computational Model suits well our on-board systems.)

Ada currently dominates the application domains which are subject to stringent normative regulations (e.g.: avionics, railway). Distinct segments of those domains bear a strong resemblance to the space domain addressed by this thesis, both process- and technology-wise. Hence, these domains should possibly develop some interest also in the enabling technology presented by this thesis.

On the other hand, C [Kernighan and Ritchie, 1988] and C++ [ISO, 1997] pervade the application domains which are more lucrative, more commercially-aggressive and with less regulated implementation standards (e.g.: automobile, telecomms). Java [Nilsen, 1996b] is a fresh new entry in the list of these and its arrival may trigger an important transformation of the market and application scenario. Many of the applications which Java is intended to serve, in fact, possess embedded real-time characteristics, even if in addition to distinctly dynamic connotations [Nilsen, 1996a].

One distinguishing characteristic of Java is its execution model. Java programs are first translated into a fully portable standard byte code representation which may then be executed on any
Java virtual machine. The appeal of the Java virtual machine concept, which simply denotes any software system capable of understanding and executing the standard Java byte code representation, lies in that it facilitates the port of Java applications and eliminates the need for cross-compiler development systems. However, as Nilsen [Nilsen, 1996a] notes, the current version and present implementations of the language also exhibit a number of important shortcomings which prevent direct use of Java in critical embedded real-time applications. The major shortcomings include: the unpredictable execution of garbage collection services (which Java implementations need to accommodate the dynamic workloads the language is designed for); the insufficient precision of task delays; the risk of priority inversion incurred on the use of Java standard libraries with unknown implementation.

The typical connotation of these market-driven technologies is to be impatient of stringent engineering discipline, to reject regulatory standards which dictate the choice of the implementation means and to antagonise Ada as the typifying expression of such impositions. This strong connotation often marries with the commonly-held perception that programming languages like C and C++ (and, possibly, Java) offer a higher degree of implementation freedom. Obviously, there is great tension between implementation freedom and adherence to a specific Computational Model. Our engineering approach clearly demands the latter. And, notably, our Computational Model can easily be implemented, for example, in C upon a POSIX-compliant platform [Gallmeister, 1995], while the relevant bindings to HRT-HOOD are not difficult to build. Examples of such bindings are presented in [E2S, 1997].

The increasing rate of software failures recently observed on board new-generation space systems (e.g.: Clementine, Pathfinder) however, seems to denote a too liberal interpretation of the "cheaper, faster, better" paradigm, all too often associated with a less rigorous development practice, perhaps on the assumption that the use of commercial-off-the-shelf software components (notably the operating system) be a sufficient assurance to success. We know all too well that this is not the case. We also notice that, in this particular respect, Ada has the important cultural and structural advantage of lending itself more naturally to the application of rigorous engineering discipline and of being self-sufficient for the implementation of embedded real-time systems.

In conclusion, we defend the effectiveness of our technology choices but also confidently look forward to seeing our engineering approach applied with foreign technology.

7.3.2 Future Work in the Space Domain

Further work may be necessary to ensure that the process model we presented in this thesis can, in the long run, co-exist and effectively co-operate with those development techniques which are the most likely candidate to become future reference technology for European space industry. We believe the "hottest" candidate techniques presently appear to be:

(a) formal description techniques, behavioural modelling and executable specifications, as the means to a faster and more reliable specification phase;

(b) automated code generation, as the means to a faster production phase;
(c) aggressive software reuse, as the means to increase the productivity of the process and the reliability of the product.

All of these techniques attempt to respond to the demand to enable the user to place more justified reliance on the dependability (hence, predictability, reliability and safety) of software-based critical systems. The strategic importance of this demand has constantly increased over the last two decades and triggered a large amount of research work in the area of validation and design for testability of software-based critical systems, including those embedded and real-time (cf., for example, [Williams, 1983], [Rushby, 1993] and [Littlewood and Strigini, 1995]). This notwithstanding, the argument developed in this thesis shows that fully automated verification environments which cover the entire system development process, from specification to execution, place far beyond the present state of the art, at least in the domain of our immediate concern.

Our strategy for this thesis work was to devise an evolutionary process model with the support of an associated enabling technology. Yet, the unabridged utilisation of any of the cited techniques is admittedly revolutionary [Fenton, 1993], hence, more inclined to "determine" the process than capable of augmenting it. The advent of these techniques in the industrial practice will thus challenge the durability of our engineering approach unless work is done to investigate how the two can be brought to an evolutionary and effective co-operation. Future work in this respect should primarily concentrate on the following two questions:

Q1. In what way can an HRT-HOOD-based design process be productively accompanied by the use of formal description techniques.

The term "formal description techniques" (FDTs) is used here to denote the application of mathematical logic and notations to the specification and verification of system design. There is no solid consensus as to which phases of the software development process are most effectively supported by FDTs. For example, [Strigini, 1993] suggests that FDTs are expected to play an important role in the capture and analysis of the user requirements as well as, in later phases of development, for proving the consistency of the system specification across successive levels of refinement. From another perspective, [Rushby, 1993] suggests the use of FDTs for the proof of correctness of a limited number of critical components of the system (like, for example, the voting algorithm of a modular-redundant system). We sympathise for the latter view, especially because of the lesser impact on the host process model. In particular, we believe that, wherever our approach is adopted, HRT-HOOD should remain the guiding notation for the establishment of the system design. This provision would in fact ensure continued and consistent control over the real-time aspects of the system. In this context, FDTs would have a productive role for the specification (and possibly production) of selected components of the system which possess characteristics that fit the notation and "world model" of the used FDT.

Q2. In what way can the structural code generation capability of HRT-HOOD co-operate with the automated code generation capability of formal description techniques and the aggressive exploitation of software reuse.
The key issue in this respect is to ensure that all of the code components which make up the overall system ultimately fit the chosen Computational Model and permit the proper real-time definition and verification of the system, irrespective of their originating production chain.

Early work in both directions has been performed within activities promoted by the European Space Agency. Interesting yet preliminary results on the practicality of productively combining reuse-oriented design with (1) HRT-HOOD as guiding notation and (2) FDT as support notation have, for example, been obtained by [CRI, 1997].
Appendix A

Glossary

AD  ARCHITECTURAL DESIGN:
phase of the software development life cycle as seen by ESA's standard practice
(descending branch of the V model).

AOC  ATTITUDE AND ORBIT CONTROL:
term denoting the on-board sub-system in charge of the attitude determination and
orbital control of the satellite.

ASATC  ASYNCHRONOUS ATC:
HOOD term designating an asynchronous request for an ATC event to occur
in a server object.

ASER  ASYNCHRONOUS EXECUTION REQUEST:
HOOD term designating a service request that involves no flow synchronisation
between the client and the server.

AT  ACCEPTANCE TESTING:
final phase of the software development life cycle as seen by ESA's standard
practice.
(ascending branch of the V model).

ATAC  ADA TASKING COPROCESSOR:
memory-mapped hardware device performing the Ada 83 tasking operations
on behalf of the Ada run-time support system.

ATC  ASYNCHRONOUS TRANSFER OF CONTROL.

COTS  COMMERCIAL OFF THE SHELF:
term designating software components of commercial origin forming part of
mission-specific software systems.

DD  DETAILED DESIGN:
phase of the software development life cycle as seen by ESA's standard practice
(ascending branch of the V model).
DHC  DATA HANDLING CONTROL:
term denoting the on-board sub-system in charge of the communications
between the satellite system and the ground centre.

DM  DEADLINE MONOTONIC.

ERC32  EMBEDDED REAL-TIME COMPUTING CORE (32 BIT):
computing core conforming with the SPARC Instruction Set Architecture
standard (v7) for use with new-generation on-board processing units
in ESA projects.

ESA  EUROPEAN SPACE AGENCY.

ESF  EXECUTION SKELETON FILE:
stylised description of the WCET profile of the threaded objects of a
real-time system.

FDT  FORMAL DESCRIPTION TECHNIQUE:
term designating the application of mathematical logic and notations
to the specification and verification of design.

HOOD  HIERARCHICAL OBJECT ORIENTED DESIGN:
the design method most frequently practiced in European space software
developments.

HRT-HOOD  HARD REAL-TIME HOOD:
an extension of the base HOOD method especially aimed at the design
of time-critical systems.

ICDS  INTEGRATED CONTROL AND DATA SYSTEM:
on-board architecture which integrates the DHC and AOC functions on a
single processing unit.

IT  INTEGRATION TESTING:
phase of the software development life cycle as seen by ESA's standard
practice (ascending branch of the V model).

OBCS  OBJECT CONTROL STRUCTURE:
HOOD notion encompassing the description of the control behaviour
of an object.

ODS  OBJECT DESCRIPTION SKELETON:
crucial component of the HOOD specification of an object.

OPCS  OPERATION CONTROL STRUCTURE:
HOOD notion describing the operation behaviour of an object.

PSER  PROTECTED SYNCHRONOUS EXECUTION REQUEST:
HOOD term designating a protected service request which causes the
control flow in the client to suspend until completion of the
protected operation.

SR  SOFTWARE REQUIREMENTS:
phase of the software development life cycle as seen by ESA's standard
practice (descending branch of the V model).
ST  **SYSTEM TESTING:**
phase of the software development life cycle as seen by ESA’s standard practice (ascending branch of the V model).

SVVP  **SOFTWARE VALIDATION AND VERIFICATION PLAN:**
document required by the ESA standard practice to specify the validation and verification strategy to apply to the development of a given software system.

TBD  **TO BE DEFINED:**
otation frequently used in software documents as a place-holder for an indeterminate item of specification.

TC  **TELECOMMAND:**
logical means of communication from the ground centre to the on-board system.

TM  **TELEMETRY:**
logical means of communication from the on-board system to the ground centre.

UCF  **USER CONFIGURATION FILE:**
stylised description of the real-time commitments of a real-time system.

UR  **USER REQUIREMENTS:**
starting phase of the software development life cycle as seen by ESA’s standard practice (descending branch of the V model).

UT  **UNIT TESTING:**
phase of the software development life cycle as seen by ESA’s standard practice (ascending branch of the V model).

WCCT  **WORST-CASE COMPUTATION TIME:**
term designating the total worst-case processing requirement incurred by a thread over a single activation, inclusive of the run-time overhead required for the execution of that activation.

WCET  **WORST-CASE EXECUTION TIME:**
term designating the worst-case processing requirement associated with the sequential execution component of a thread, computed over a single release.
Appendix B

Summary

B.1 Subject

An operational space system is made up of two co-operating components: a space segment and a ground segment. In the case of unmanned space systems in general and satellite systems in particular, the space segment is typically comprised of two building blocks: the platform, which performs the on-board management and control functions, and the payload, which discharges the mission-specific value-added services of the space system. Since the mid 70s, satellite builders have started to use software-controlled microprocessors for the implementation of an increasing proportion of such functions and services.

Our research work addressed the engineering of the real-time software embedded on board the platform computer of new-generation satellite systems.

The space segment operates in a prohibitive environment for conventional ground technology. On-board processing technology must in fact be resilient to radiations, heavy particles and other space effects. This technology is costly and does not develop as rapidly as the equivalent for ground application. Up to the recent past, therefore, the design decision as to where to allocate the mission-critical functions of a space system was curtailed by the poor processing capability availed to the space segment. This status of affairs is now being shaken by the concurrence of two phenomena: the recent availability of more modern and powerful on-board computers and the rise of the "cheaper, faster, better" project paradigm determined by the growing commercial interest in the exploitation of space systems. In broad terms, the new situation propels important demands for: drastic reduction of on-board hardware and harness requirements ("cheaper"); shorter time to market ("faster"); more effective and responsive utilisation of the on-board processing resources ("better") for the delivery of increased mission product. A further corollary of the rising paradigm is the progressive move of critical functions from the ground segment to the on-board platform. As a result of this evolution, new-generation on-board systems appear to be: (1) increasingly software-intensive, as software pervades an ever-growing proportion of on-board functions; (2) increasingly concurrent, as on-board processors are to perform, in parallel, a growing variety
of control activities which integrate together all of the platform functions on a single on-board computer; (3) increasingly time-critical, in that a growing proportion of the on-board software components are subject to mission-critical requirements on timeliness of execution.

B.2 Problem

The focus of our research work was placed on the deficiencies of the current development process and the limitations of conventional on-board software architectures in the face of the rising "cheaper, faster, better" project paradigm.

Our analysis resulted in two major observations:

- The present effort distribution across the software development process is excessively shifted towards the verification phase in the ascending branch. This ill-balancedness is critical, because the verification phase has an inherently centralised and sequential connotation, which can hardly accommodate the required reduction of the development schedule other than at the price of a lesser-quality product. Hence, this drawback needs to be resolved if the process is to master the increasing complexity of new-generation systems and concurrently accommodate the anticipated reduction of the development schedule.

- The design and implementation phase in the descending branch appear to be insufficiently equipped to cope with greater real-time criticality of new-generation systems. The inadequacy of the conceptual means presently availed to this phase is manifested by the conservatism of the conventional architectural approach, which employ rigid and inflexible cyclic scheduling schemes as the sole response to the real-time demands of the system. The monolithic architecture of these systems makes them ill-equipped to fit in the anticipated process scenario, expectedly characterised by incremental builds, late changes and development iterations, which will result from the concurrent engineering of the components of new-generation systems.

We opted to address these issues with a view to the applicability and cost-effectiveness of the envisaged solution to the present industrial practice in the domain. Accordingly, we strived to construct an engineering approach which would place in an evolutionary relationship to the process and technology presently in use at European space industry.

B.3 Solution

We based our strategy around two crucial provisions:

1. the introduction of a verification process centred around static real-time analysis, as a substitute for dynamic real-time testing; and

2. the adoption of an architectural paradigm based upon fixed priority preemptive scheduling, as an evolutionary amendment to the current practice geared to attain the combination of flexibility and predictability required of new-generation on-board systems.
In order to implement this strategy, we augmented the current software process by two major enhancements:

- the support for a unified *design framework* to embrace the real-time design activities in the descending branch into a single, iterative and incremental process stage; and
- the notion of *Computational Model* to augment the expressive power of the design definition phase to capture the crucial constructive elements of real-time systems.

Subsequently, we determined that the segment of the software development process covered by the design framework is to proceed across four iterative and incremental stages, as follows:

1. *real-time design*: which encompasses the definition of the logical and the physical model of the system and focuses on the characterisation of the real-time requirements placed on the individual components of the system;

2. *binding to Computational Model*: in which, the source program extracted from the current design is bound to the run-time instantiation of the Computational Model of choice and the resulting system submitted to the compilation system for the determination of the worst-case execution time profile of its application components;

3. *static analysis*: in which, response time analysis is performed statically to ascertain whether the current implementation of the system meets the real-time requirements placed on the design;

4. *feedback to design*: which is entered to review the results obtained from the earlier stage of analysis and determine how to complete the design and implementation of the system before it is submitted to the testing phase.

The final element of our engineering solution encompassed the definition, implementation and demonstration of the enabling technology needed to support our process. Coherently with the premises to our work, we built this technology as an evolutionary enhancement of the technology baseline in place at European space industry, which is presently based around the HOOD method for the design and Ada 83 for the implementation.
Appendix C

Samenvatting

C.1 Onderwerp

Een operationeel ruimtevaartysteem bestaat uit twee samenwerkende componenten: een space segment en een ground segment. In het geval van onbemande ruimtevaartystemen in het algemeen en van satellietsystemen in het bijzonder bestaat het space segment gewoonlijk uit twee bouwstenen: het platform, dat de beheer en regel functies aan boord uitoefent, en de payload die de missie specifieke diensten met de eigenlijke toegevoegde waarde van het ruimtevaart systeem bevat. Sinds het midden van de zeventiger jaren zijn satelliet bouwers begonnen met het gebruiken van software gestuurde microprocessoren voor een toenemend aantal van zulke functies en diensten.

Ons onderzoek richt zich op het ontwerpen van de real-time software voor de platform computer van de nieuwe generatie satelliet systemen.

Het ruimtevaartsegment werkt in een vijandige omgeving voor conventionele grond gebaseerde technologie. De technologie voor het verwerken van informatie aan boord moet bestand zijn tegen straling, zware deeltjes en andere ruimte effecten. Deze technologie is kostbaar en ontwikkelt zich niet zo snel als de vergelijkbare technologie voor grond gebaseerde toepassingen. Tot het recente verleden werden de ontwerp beslissingen over waar de kritieke functies voor de missie moesten worden geplaatst, gestuurd door de lage informatie verwerkings capaciteit van het ruimte segment. Deze stand van zaken wordt nu op zijn kop gezet door het samen lopen van twee verschijnselen: de recente beschikbaarheid van modernere en krachtigere vluchtcputurers en de opkomst van het "goedkoper, sneller, beter" project paradigm door de groeiende commerciële belangstelling voor het exploiteren van ruimtevaart systemen. In algemene termen, de nieuwe situatie stimuleert de vraag naar: een drastische reductie van de hoeveelheid hardware en van de bekabeling aan boord ("goedkoper"); een kortere ontwikkelijd ("sneller"); een effectiever en slagvaardiger gebruik van de aan boord aanwezige informatie verwerkings mogelijkheden ("beter") om een beter missie produkt te kunnen leveren. Verwant met dit opkomend paradigm is de groeiende tendens om kritische functies van het ground segment naar het platform van het
space segment te verhuizen. Als gevolg van deze evolutie worden de nieuwe generatie boordsystemen: (1) in toenemende mate software intensief, omdat software deel is gaan uitmaken van een groeiend aantal functies aan boord; (2) in toenemende mate concurrent, omdat de processoren aan boord een groeiend aantal beheers en regel activiteiten in parallel moeten uitvoeren, die alle platform functies integreren op een enkele vluchtcomputer; (3) in toenemende mate time critical, omdat een groeiend aandeel van de software componenten aan boord moeten voldoen aan missie kritische vereisten op het gebied van een tijdige uitvoering.

C.2 Probleem

Ons onderzoekwerk heeft zich geconcentreerd op de zwakheden van het huidige ontwikkelproces en de beperkingen van conventionele software architecturen voor gebruik aan boord in het licht van het opkommende "goedkoper, sneller, beter" project paradigma.

Onze analyse resulteerde in twee belangrijke waarnemingen:

- In de huidige verdeling van werk in het software ontwikkelproces heeft de verificatiefase in de stijgende tak een bovenmatig aandeel. Deze onevenwichtigheid is kritiek omdat de verificatiefase altijd een inherent gecentraliseerd en sequentieel element heeft, dat nauwelijks aan de eis van een verkorting van de ontwikkeltijd kan voldoen tenzij door een product van lagere kwaliteit af te leveren. Dit probleem dient opgelost te worden indien het proces in staat moet zijn om tegelijkertijd de toenemende complexiteit van de nieuwe generatie systemen en de verwachte verkorting van de ontwikkelduur aan te kunnen.

- De ontwerp en implementatiefase in de dalende tak zijn onvoldoende uitgerust om het toenemend real-time kritisch zijn van de nieuwe generatie systemen te kunnen behandelen. De ondoelstrekkendheid van de conceptuele middelen die nu voor deze fase beschikbaar zijn wordt geïllustreerd door het conservatisme van de conventionele architecturale benadering die gebruik maakt van vaste en inflexible cyclische schedule schemas als het enige antwoord op de real-time eisen van het systeem. De monolitische architectuur van deze systemen maken hun slecht uitgerust om te passen in het verwachtte processenario dat gekarakteriseerd wordt door inkrementele versies, late veranderingen in de eisen en ontwikkel iteraties die het gevolg zijn van het concurrent ontwerpen van de componenten van de nieuwe generatie systemen.

We hebben gekozen om deze problemen te behandelen met nadruk op de toepasbaarheid en de kosten/baten verhouding van de voorziene oplossing op de huidige industriële praktijk in het domein. In overeenstemming hiermee hebben wij gestreefd om een ontwerp benadering te volgen die een evolutie is van het proces en de technologie die nu in gebruik zijn bij de Europese ruimtevaartindustrie.

C.3 Oplossing

We hebben onze strategie gebaseerd op twee cruciale elementen:
1. de introductie van een verificatieproces gecentreerd rond statische real-time analyse als een vervanging voor dynamisch real-time testen; en

2. de adoptie van een architectureel paradigm gebaseerd op vaste prioriteit preemptive scheduling als een evolutionele amendement op de huidige praktijk gericht op het bereiken van de combinatie van flexibiliteit en voorspelbaarheid die door de nieuwe generatie vluchtsystemen wordt vereist.

Om deze strategie te implementeren hebben wij aan het huidige software proces twee belangrijk verbeteringen toegevoegd:

- de ondersteuning voor een algemeen design framework dat de real-time ontwerpactiviteiten in de dalende tak samen brengt tot een enkele, iteratieve en incrementele proces stap; en

- de notie van een computational model om het uitdrukkingenvermogen van de ontwerp definitie fase zo te laten toenemen dat het de cruciale bouwelementen van real-time systemen kan bevatten.

Vervolgens hebben wij bepaald dat het segment van het software ontwikkelproces dat door het design framework wordt bestreken dient te bestaan uit vier iteratieve en incrementele stappen en wel als volgt:

1. real-time ontwerp: deze stap omvat de definitie van het logische en fysische model van het systeem en richt zich op het karakteriseren van de real-time eisen die aan de afzonderlijke componenten van het systeem worden gesteld;

2. het verbinden aan een computational model: in deze stap wordt het bron programma uit het huidige ontwerp verbonden met de run-time instantiatie van het computational model naar keuze en het resulterende systeem wordt overgegeven aan het compilatiesysteem voor de bepaling van het uitvoeringstijdprijsel van de applicatie componenten onder de slechtst denkbare omstandigheden;

3. statische analyse: in deze stap wordt een statische responsietijd analyse gedaan om te bepalen of de huidige systeem implementatie voldoet aan de real-time eisen die aan het ontwerp gesteld zijn;

4. terugkoppeling naar het ontwerp: in deze stap worden de resultaten bekeken, die verkregen zijn uit de eerdere fase van de analyse, om te bepalen hoe het ontwerp en de implementatie van het systeem voltooid moeten worden voordat het systeem overgaat naar de testfase.

Het laatste element van onze ontwerp oplossing omvat de definitie, implementatie en demonstratie van de technologie die nodig is om ons proces te ondersteunen. In overeenstemming met de uitgangspunten voor ons werk hebben wij deze technologie gebouwd als een evolutionele verbetering van de nu in gebruik zijnde technologie bij de Europese ruimtevaart industrie die is gebaseerd op de HOOD methode voor het ontwerp en Ada83 voor de implementatie.
Appendix D

Curriculum Vitae

Tullio Vardonega was born on the 12th of January 1963 at La Spezia, Italy. He graduated with full marks in Computer Science in July 1986 at the Computer Science Department of the University of Pisa, Italy. He served one and a half year civilian service in a social solidarity organisation, which stands out as one of his key accomplishments to date. He joined Intecs Sistemi at their site at Pisa in November 1987. At Intecs Sistemi, he was team leader in two international co-operative research projects which focused on the design and implementation of real-time concurrent applications. He joined the European Space Agency in July 1991, as staff member of the Agency's Research and Technology Centre based at Noordwijk, The Netherlands. At the Agency, he specialised in on-board real-time embedded computing and was the initiator and coordinator of several research projects in that domain. In February 1996 he approached Professor J. van Katwijk at the Technical University of Delft, The Netherlands, with a proposal for an external doctoral thesis centred around the elaboration of the major results obtained from those projects. The proposal was accepted. This document is the result of that effort.
Bibliography


