Development of On-Board Embedded Real-Time Systems: An Engineering Approach

TULLIO VARDANEGA

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

1st edition (August 1998):
As doctoral thesis presented at the Technical University of Delft (NL).
Approved on October 6, 1998 by the following exam board:
Prof. O.J. Olsder, Technical University of Delft, chairman.
Prof. J. van Katwijk, Technical University of Delft, promotor.
Prof. A. Burns, University of York, (UK).
Prof. H. Koppelaar, Technical University of Delft.
Prof. J. Zalewski, University of Central Florida (USA).
Prof. J.-A. de la Puente, Technical University of Madrid (E).
Prof. H. Sips, Technical University of Delft
for Prof. F. Panzieri, University of Bologna (I).
Dr. W. Toetenel, Technical University of Delft
for Prof. K. De Vlaminck, Catholic University of Leuven (B).

2nd edition (November 1998):
As technical report for publication by the European Space Agency
Research and Technology Centre (ESTEC) at Noordwijk (NL).

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 SR 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.

It is apparent that, in the face of the anticipated reduction of the development schedule, the level of effort devoted to the testing phase cannot increase proportionally with the 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 ill-balancedness of the present effort distribution across the 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* (introduced in section 3.4.4), 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*, to embrace the activities of the descending branch (i.e. requirements analysis; design definition; design analysis and implementation) into a single, unfractured development phase within which they can proceed in an *iterative* and *incremental* fashion and ordinately respond to the *feedback* arising from the progressive consolidation of the system.

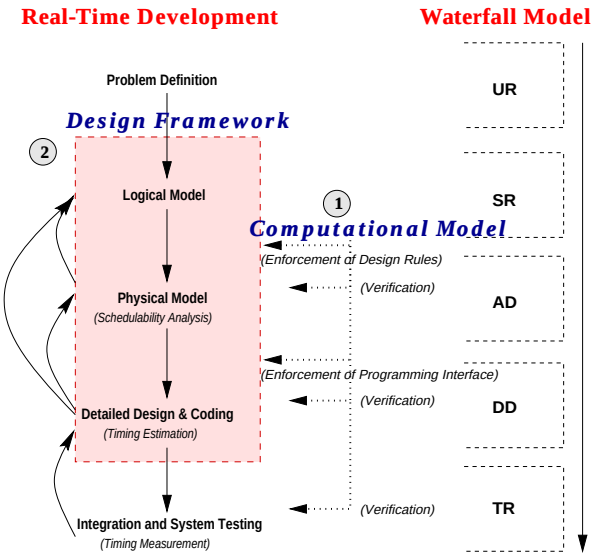


Figure 4.1: The Proposed Software Process.

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

The basic rationale of our development approach 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. in the cited paper 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. argue that the construction of real-time application software which does not consider the characteristics of the target system is totally inadequate. The cited authors also insist 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. maintain that time must be represented in a *uniform and concise* way all along the development process. Finally, the cited authors advocate the need for means capable of supporting the *early evaluation* of the design with respect to its timeliness requirements.

In fact, our work departs from that of Fohler et al. in the respect of technical choices made to respond to the same core of identified needs. Kopetz in [Kopetz et al., 1993] outlines the implementation approach taken by the MARS team in accord with the cited work of Fohler et al. In contrast with that approach, our work acknowledges the occurrence of feedback-based development iterations across a greater portion of the process than the Kopetz approach allows and builds on them for the incremental consolidation of the system.

Our approach has also much in common with some of the views expressed by Kruchten in [Kruchten, 1995]. According to Kruchten, common experience has it 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. 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 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 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 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.

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. We contend, however, 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. We argue that this provision simplifies the transitions and iterations between the two.

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 practicable enhancements to 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 form an overall picture 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 technical report.

Table 4.1: Overview of Driving Requirements.

Required Property	Justification	Origin
C.1	assertion **A.2.1**	sect. 1.4 & sect. 3.3.1
C.2	assertion **A.1.1-3**	sect. 1.4 & sect. 3.3.2
C.3	requirement **R.1, R.3**	sect. 1.4
C.4	strategic decision	sect. 3.3

This technical report 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 technical report (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 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 our 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;

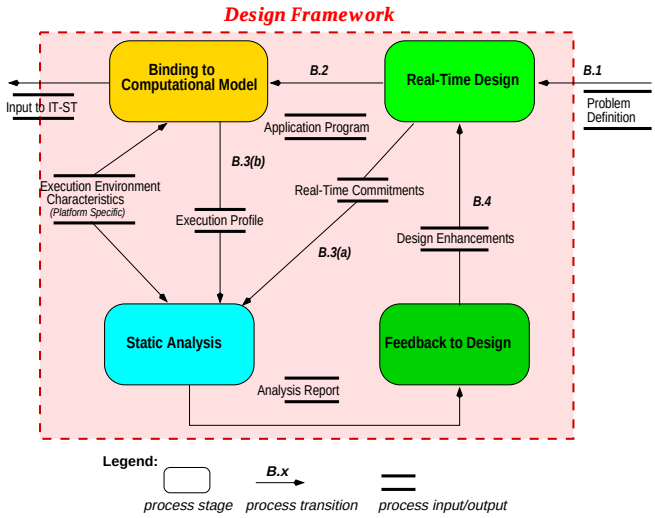


Figure 4.2: Design Activities in the Proposed Process.

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 execution 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 inadequate 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.

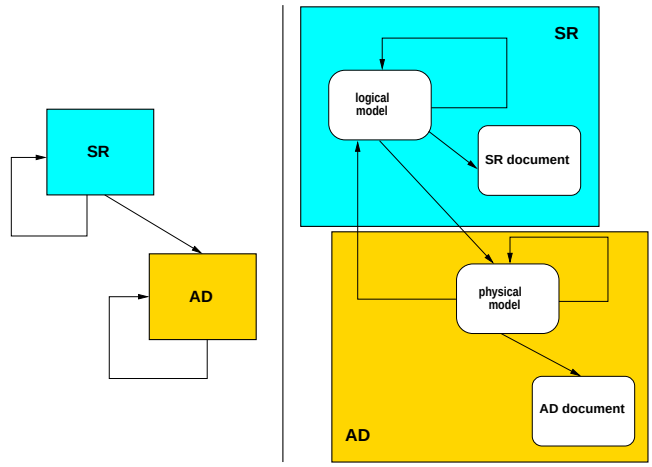


Figure 4.3: Feedback-Based Iterative Design Flow.

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 technical report 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 technical report 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 on-board 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 preemptive 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 technical report 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 technical report.

It is not the intent of this technical report 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 (similarly to 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 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 4.2: Disallowed *"Implemented By"* Relationships.

CHILD / PARENT	**cyclic**	**sporadic**	**protected**	**passive**
cyclic				
sporadic				
protected	X	X		
passive	X	X	X	

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.

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 feed-

Table 4.3: Allowed & Disallowed *"Use"* Relationships.

USED / USER	**cyclic**	**sporadic**	**protected**	**passive**
cyclic				
sporadic				
protected	ASATC	ASATC, START	unconstrained	
passive	X	X	X	

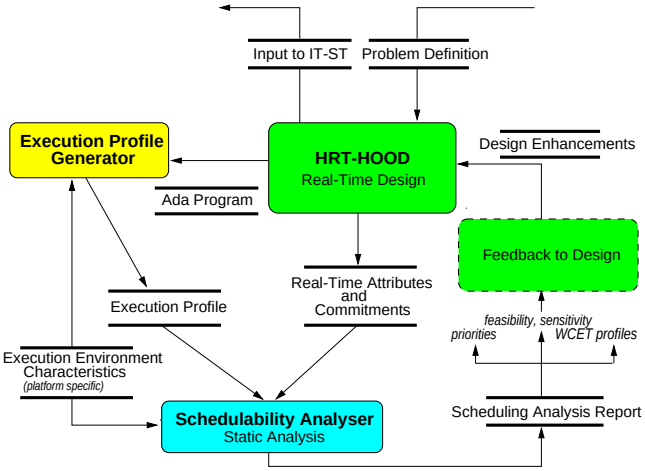


Figure 4.4: Design Activities in Operation.

back. 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 [Intecs 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:

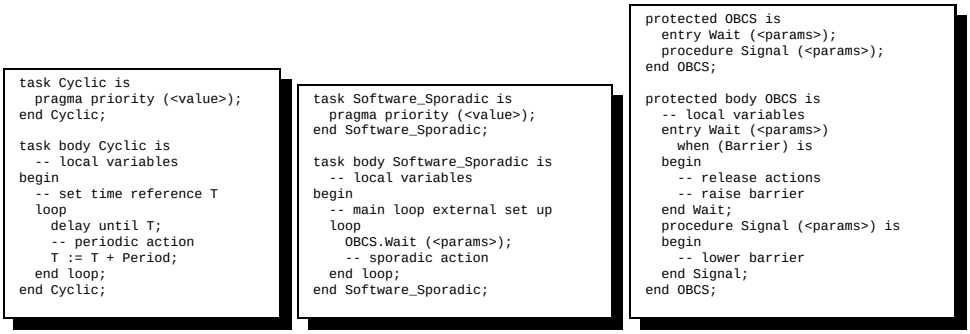
- the *thread*, which is to perform the time-triggered operation of the object and maps to an independent thread of control implemented as an *entry-less* Ada task with a single, time-triggered invocation event and a potentially unbounded number of invocations; and
- the *OBCS*, which is needed only 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 along with its OBCS (synchronisation server).



```
task Cyclic is
  pragma priority (<value>);
end Cyclic;

task body Cyclic is
  -- local variables
begin
  -- set time reference T
  loop
    delay until T;
    -- periodic action
    T := T + Period;
  end loop;
end Cyclic;
```

```
task Software_Sporadic is
  pragma priority (<value>);
end Software_Sporadic;

task body Software_Sporadic is
  -- local variables
begin
  -- main loop external set up
  loop
    OBCS.Wait (<params>);
    -- sporadic action
  end loop;
end Software_Sporadic;
```

```
protected OBCS is
  entry Wait (<params>);
  procedure Signal (<params>);
end OBCS;

protected body OBCS is
  -- local variables
  entry Wait (<params>)
    when (Barrier) is
  begin
    -- release actions
    -- raise barrier
  end Wait;
  procedure Signal (<params>) is
  begin
    -- lower barrier
  end Signal;
end OBCS;
```

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 *no-cache no-virtual memory* single-board computers for advanced new-generation on-board systems. Amongst other features, which are not discussed in this technical report, the ERC32 chipset and associated Ada compilation system optionally supports board configurations which include the ATAC (Ada Tasking Coprocessor) chip [Roos and Gomez-Molinero, 1992]. The ATAC is a memory-mapped hardware device which performs Ada 83 tasking operations on behalf of classical software run-time 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 pessimistic 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 characteristics file*.

Table 4.4: Timing Characteristics of Basic Run-Time System Primitives (μs).

RTS Primitive	**Used by / for**	**DEM32 (10 MHz 0 Wait-states)**	
		non-ATAC run-time	**ATAC run-time**
PO_Entry	cyclic, sporadic, interrupt	8.0	13.8
PO_Exit	cyclic, sporadic, interrupt	11.0	11.0
Int_Wait (Enter)	interrupt	3.0	3.0
Int_Wait (Exit)	interrupt	3.0	0.0
Sem.Wait (Enter)	sporadic	7.0	7.0
Sem.Wait (Exit)	sporadic	3.0	3.0
Entry_Queue_Mgmt	sporadic	6.0	8.0
Select	cyclic, sporadic, interrupt	5.0	0.0
Switch	cyclic, sporadic, interrupt	34.0	41.0
Delay Until (Insert at Top)	cyclic	39.0	23.0
Delay Until (Insert Lower)	cyclic	$22.0 + \mathbf{C} * 3.0$	23.0
Delay Until (Exit)	cyclic	8.0	8.0
Timer_Int	cyclic	21.0	0.0
Ready	cyclic	12.0	0.0
Int_Handling	interrupt	67.0	0.0
Max_Deferred_Preemption		130.0	65.0

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 operation; as it will be explained in section 4.5.3, in fact, that value contributes to the determination 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 version of the system, this value remains constant as the thread queuing is taken care of entirely by the ATAC chip itself.

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 τ_i does, thus, result from suitable combination of the following three distinct components:

(i) The *worst-case computation time* of thread τ_i, $\mathbf{WCCT}_i$, which is defined as the sum of the time cost of all τ_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 τ_i, $\mathbf{I}_i$, which is caused by the occurrence of preemptive execution of higher-priority threads and higher-priority run-time system services incurred during τ_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 τ_i, $\mathbf{B}_i$, which originates from the possibility that a due release of τ_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 τ_i, component $\mathbf{WCCT}_i$ is fully determined at compile time on the Ada closure of the program, component $\mathbf{B}_i$ is a function of the assigned priorities and the system's run-time performance, and component $\mathbf{I}_i$ is a function of the system load.

Component $\mathbf{WCCT}_i$ and $\mathbf{B}_i$ 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 $\mathbf{B}_i$ and the issues in the generation of the worst-case execution profiles from which $\mathbf{WCCT}_i$ is determined are discussed separately in the following.

Component $\mathbf{I}_i$ 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 τ_i's ready period over the interval $[0, t)$ are shown in the following, where notation $j \epsilon HP(i)$ denotes that thread τ_j's priority is greater than τ_i's (i.e. $Pr(\tau_j) \geq Pr(\tau_i)$) notation $j \epsilon LP(i)$ denotes the converse (i.e. $Pr(\tau_j) < Pr(\tau_i)$) and T_j denotes τ_j's period (for cyclic threads) or minimum interarrival time (for sporadic threads):

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

$$\mathbf{I}_i^t = \sum_{j \epsilon HP(i)} \lceil \frac{t}{T_j} \rceil \mathbf{WCCT}_j \tag{4.1}$$

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

$$\mathbf{C}_i^t = \mathrm{K}_i^t * (Timer\_Int + Ready + Select) \tag{4.2}$$

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 the interrupts off the Interval Timer and *only* releases those which induce preemption:

$$\mathrm{K}_i^t = \sum_{j \epsilon HP_{CYCLIC}(i)} \lceil \frac{t}{T_j} \rceil \tag{4.3}$$

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:

$$\mathrm{K}_i^t = \sum_{j \epsilon HP_{CYCLIC}(i)} \lceil \frac{t}{T_j} \rceil + \sum_{k \epsilon LP_{CYCLIC}(i)} 1 \tag{4.4}$$

No further releases of lower-priority threads may occur before τ_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 τ_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) :

$$\mathrm{R}_i^n = \mathbf{B}_i + \mathbf{WCCT}_i + \mathbf{I}_i^{\mathrm{R}_i^{n-1}} + \mathbf{C}_i^{\mathrm{R}_i^{n-1}} \qquad (n > 1) \tag{4.5}$$

$$\mathrm{R}_i^1 = \mathbf{B}_i + \mathbf{WCCT}_i \tag{4.6}$$

Response Time (AD) :

Busy Window at (q+1)th release:

$$\mathrm{R}_i^{n+1}(q) = \mathbf{B}_i + (q+1)\mathbf{WCCT}_i + \mathbf{I}_i^{\mathrm{R}_i^n(q)} + \mathbf{C}_i^{\mathrm{R}_i^n(q)} \qquad (q, n \geq 0) \tag{4.7}$$

where:

$$\mathrm{R}_i^0(q) = \mathrm{R}_i(q-1) \qquad (q \geq 1) \tag{4.8}$$

and

$$\mathrm{R}_i^0(0) = \mathbf{B}_i + \mathbf{WCCT}_i \tag{4.9}$$

Response Time at (q+1)th release:

$$\mathrm{R}_i(q) = \mathrm{R}_i^n(q) - qT_i \tag{4.10}$$

Worst-Case Response Time:

$$\mathrm{R}_i = \max_{q \epsilon N} \mathrm{R}_i(q) \tag{4.11}$$

where:

$$N = \{q\} : \mathrm{R}_i^n(q) > (q+1)T_i \Longrightarrow \mathrm{R}_i(q) > T_i \tag{4.12}$$

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-ceiling 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 4.5: Types of PO Calls.

server type	call type	call denotation
resource server (RPO)	unguarded service call	RPO.Service
synchronisation server (SPO)	unguarded releasing call	SPO.Signal
synchronisation server (SPO)	guarded suspensive call	SPO.Wait

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 $\mathbf{WCCT}$ 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 4.6: PO Call Overhead.

call type	execution components	id	case type
RPO.Service	*PO_Entry* + **WCCT**(Service) + *PO_Exit*	(1)	worst,best
SPO.Signal	*PO_Entry* + **WCCT**(Signal) + + *Entry_Queue_Mgmt* + *Switch*	(2)	worst
SPO.Wait (in)	*PO_Entry* + **WCCT**(guard_eval) + + *Sem.Wait (Enter)* + *Select* + *Switch*	(3.1)	worst
SPO.Wait (out)	*Sem.Wait (Exit)* + **WCCT**(guard_eval) + + **WCCT**(Wait) + *PO_Exit*	(3.2)	worst
SPO.Signal (no_wait)	*PO_Entry* + **WCCT**(Signal) + *PO_Exit*	(4)	best
SPO.Wait (open)	*PO_Entry* + **WCCT**(guard_eval) + + **WCCT**(Wait) + *PO_Exit*	(5)	best

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. $w \epsilon HP(s)$, the execution cost of (3.2) is captured by the interference incurred by the signaller on its worst-case execution ($\mathbf{I}_s$);

- 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 $w \epsilon LP(s)$; 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 4.7: Call Overheads Accountable for Blocking.

	call type			**blocking factor**	
	RPO.Service	**SPO.Signal**	**SPO.Wait**	**Worst Case**	**Best Case**
Pr(Caller) vs Pr(τ_i)	H	H	H	none	none
	L	H	H	(1)	(1)
	L	L	H	max(1,2)	max(1,4)
	L	L	L	max(1,(2+3.2),(3.1))	max(1,4,5)
	H	L	L	max((2+3.2),(3.1))	max(4,5)
	H	H	L	(3.1)+(3.2)	(5)
	H	L	H	(2)	(4)
	L	H	L	max(1,((3.1)+(3.2)))	max(1,5)

The information in table 4.7 is to be interpreted as follows:

let J denote the thread set captured by equation 4.13:

$$\{\tau_j\}_{j \epsilon J} : (j \neq i) \wedge (j \epsilon LP(i)) \wedge (j \Longrightarrow PO.\mathrm{E}) \wedge (PO \epsilon HP(i)) \tag{4.13}$$

where the notation $j \Longrightarrow PO.\mathrm{E}$ denotes that τ_j's WCET profile includes a PO call (PO.E) of the type indicated by the relevant column header in the table;

with these provisions, entries tagged L in columns 2-4 denote the fact that:

$$\exists k : k \epsilon J \wedge (\mathbf{WCCT}_k(\mathrm{E}) = max_{j \epsilon J}(\mathbf{WCCT}_j(\mathrm{E}))) \tag{4.14}$$

whereby the PO.E call made by τ_k represents the largest priority ceiling blocking caused on τ_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, τ_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 $\mathbf{WCCT}$ 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 4.8: Task Administration Overhead Bounds.

task / event	**analysis case**	
	DM & AD ($q = 0$)	**AD ($q > 0$)**
Cyclic		
on release	Switch + Delay Until(Exit)	0
on suspension	Delay Until(Enter) + Select + Switch	0
Int Sporadic		
on release	Int_Handling + Select + Switch + Int_Wait(Exit)	idem
on suspension	Int_Wait(Enter) + Select + Switch	idem
Sw Sporadic		
on release	SPO.Wait (out) *if* $w \epsilon HP(s)$	idem
	0 *if* $w \epsilon LP(s)$	idem
on suspension	SPO.Wait (in)	SPO.Wait (in) *if* $w \epsilon HP(s)$
		SPO.Wait (open) *if* $w \epsilon HP(s)$
Any Task Type		
RPO.Service	as per table 4.6	idem
SPO.Signal	SPO.Signal(no_wait) *if* $w \epsilon HP(s)$	SPO.Signal(no_wait)
	SPO.Signal + SPO.Wait(out) + Select + Switch *if* $w \epsilon LP(s)$	
Interval Timer		
on cyclic release	$\mathbf{K}_i^t$ * (Timer_Int + Ready + Select)	idem

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 $\tau_i : w \epsilon HP(\tau_i)$; and
- kicked off its entry queue by its respective signaller, for every signaller task $s : w \epsilon LP(s)$.

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 $\mathbf{WCCT}$ component for that thread. As indicated earlier in section 4.5.3, thread τ_i's $\mathbf{WCCT}_i$ 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 τ_i's main loop; and (ii) the run-time system overhead incurred by τ_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, taking advantage of the fact that the designated target hardware is a single-board *cache-less* computer based on the ERC32 core mentioned earlier in this paper. In fact, the present version of the WCET tool was primarily designed to serve as a "concept demonstrator"; role which it has successfully accomplished. Our future plan is to thoroughly review the performance of the tool (particularly as regards the control of pessimism) and push it further 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) **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) **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 $\mathbf{WCCT}_i$ is not the sole objective of the WCET profile generator. There, in fact, exist two distinct ways for thread τ_i to effect system's responsiveness:

- a longer $\mathbf{WCCT_i}$ induces a longer $\mathbf{I}_j$ on thread $\tau_\mathbf{j} \forall j : j \epsilon LP(i)$;
- a PO.Call performed by τ_i may contribute to $\mathbf{B}_k$ for thread τ_k $\forall k : k \epsilon HP(i) \wedge k \epsilon LP(PO)$.

The WCET profile generation algorithm must, therefore, also seek to achieve justified maximisation of $\mathbf{B}_k$.

```
if <condition> then
   -- branch A
   <A1> -- sequential block
   PO.Call
   <A2> -- sequential block
else
   -- branch B
   <B>  -- sequential block
end if;
```

Example 4.5.1: Effect of Blocking on Path Selection.

Consider the code fragment in example 4.5.1 and assume that $< condition >$ cannot be statically resolved. The example shows a classical case in which *local* maximisation of $\mathbf{WCCT}_i$

may degrade the determination of $\mathbf{B}_k$ for any thread τ_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* blocking analysis be performed of the potential blocking effect of such calls. This analysis may possibly yield a larger $\mathbf{B}_j$ value for some thread τ_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 τ_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 deter-

mined for that thread and its $\mathbf{WCCT}$ and $\mathbf{B}$ 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 $\mathbf{WCCT}$ 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 technical report.

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

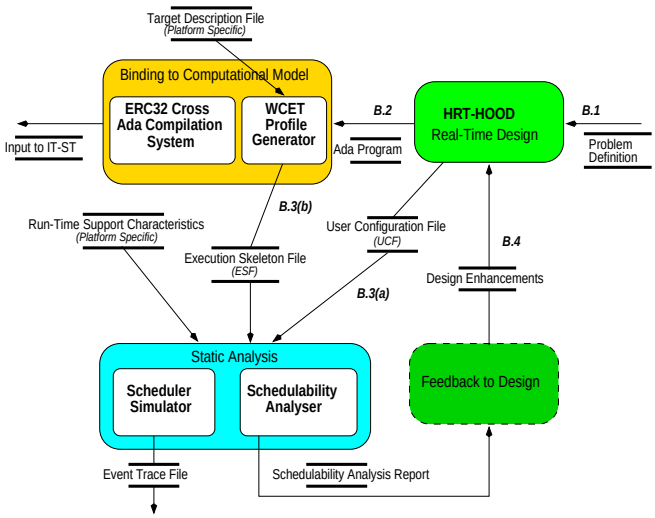


Figure 4.6: The Proposed Toolset.

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.

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

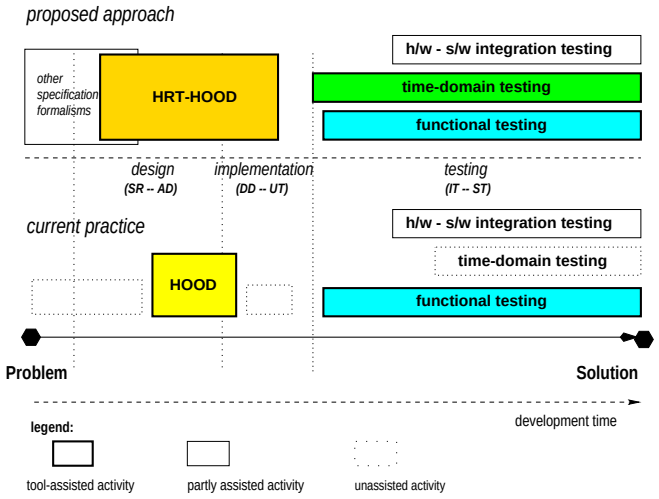


Figure 4.7: A Schematic View of the Proposed Approach.

comprised of SR and AD and typically represents 25-30% of the development; *implementation*, which corresponds to DD and UT and typically absorbs $\pm$ 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 technical report, 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 technical report.

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 described 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, but syntactically different, to 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 facilitate the definition, the extraction and the 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 retained as the centre of our engineering approach is based on the deadline monotonic scheduling theory established by [Audsley et al., 1991] and [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 fulfil 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 relationships. 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 cater 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

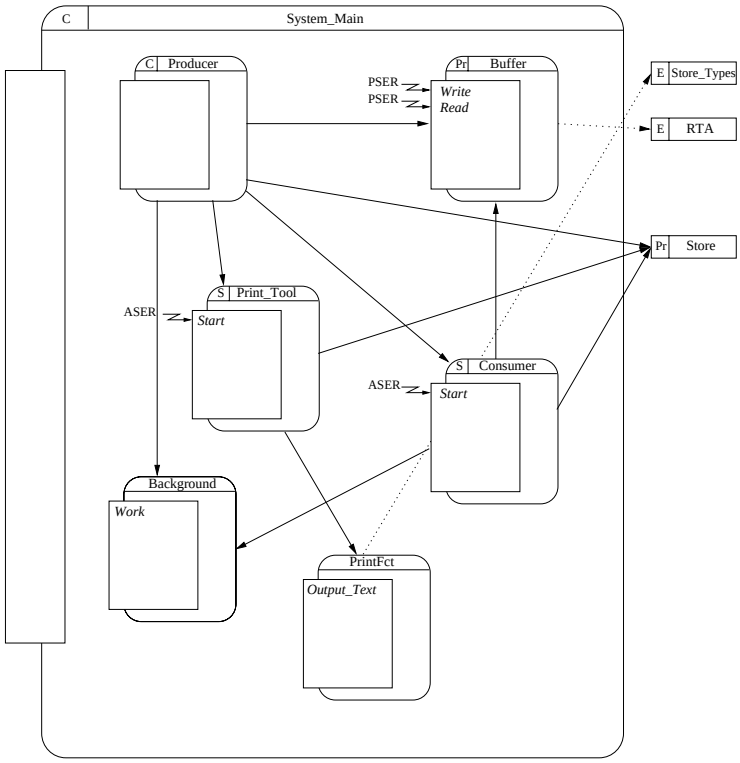


Figure 5.1: A Simple HRT-HOOD System.

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 **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 *Start* operation required to trigger the sporadic operation of the object. As required by the method, both instances of *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) *Read* and *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 Write operation of the latter;
- performs a given amount of internal work using the Work operation provided by the Background object;
- records in Store, using the 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 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 *Write* operation provided by the Buffer protected object;
- the unconstrained *Work* operation provided by the Background passive object;
- the asynchronous *Start* operation provided by the Consumer sporadic object;
- the protected synchronous *Write* operation provided by the Store protected object (not visible in figure 5.1);
- the asynchronous *Start* operation provided by the Print_Tool sporadic object.

Producer, Consumer and Print_Tool are *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, say, 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 (denominated 'importance' by the tool in use);
- 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.

```
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 translated 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 terminal object in the system (passive objects have no real-time properties).

```
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.

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

```
--| "with" preamble
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-HOOD design.*

As mentioned in section 5.2, the most prominent feature of an HRT-HOOD 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 contention 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 deferred 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 scheduling. 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.Obcs.Read(Item);
  Store.Obcs.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 discused 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_RTATT.INITIAL_PRIORITY);
 end THREAD; -- from HRT attributes
 task body THREAD is
   Start_BUFFER : Start_PARAMETER_SET;
 begin
   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 Computational 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 OBCS is
   pragma PRIORITY(Buffer_RTATT.INITIAL_CEILING);
                  -- from HRT attributes Buffer
   pragma Passive;
   entry Write(Item : in Integer);
   entry Read(Item : out Integer);
 end OBCS;
end Buffer;
```

Figure 5.11: Buffer (spec).

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

```
PROTECTED <Server_Ada_Name>
 TYPE SYNCHRO
 -- OBCS have one barriered entry
  BARRIER WCET Cp, Cmr, Cmw   -- worst-case time for the
                              -- evaluation of the barrier
  ENTRY <Barriered_Entry_Name> -- only one
  -- worst-case execution profile of barriered 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 he 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 $< Cp, Cmr, Cmw >$ 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 $\mathbf{WCCT}$ 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
  THREAD PRODUCER.THREAD
    TYPE CYCLIC
     CALL_PO BUFFER.OBCS WRITE
     WCET 50570, 12614, 4211
     CALL_PO CONSUMER.OBCS START
     CALL_PO STORE.OBCS WRITE
     CALL_PO PRINT_TOOL.OBCS START
  END
  THREAD CONSUMER.THREAD
    TYPE SPORADIC
     CALL_PO CONSUMER.OBCS WAIT_START
     CALL_PO BUFFER.OBCS READ
     CALL_PO STORE.OBCS WRITE
     WCET 50490, 12606, 4210
  END
  PROTECTED CONSUMER.OBCS
    TYPE SYNCHRO
     ENTRY START      WCET 35, 2, 1
     BARRIER WCET 6, 1, 0
     ENTRY WAIT_START WCET 38, 5, 5
  END
  PROTECTED BUFFER.OBCS
    TYPE RESOURCE
     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) can 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).

```
List of all schedulable and unschedulable threads:
```

Sch	Th#	Crit	Deadline	Prio	WCCT	Period MIAT	Response Time	Blocking Time	Blocking Origin	Margin Analysis
yes	3	HARD	8000	9	160	20000	428	130	RTS	1.00E+03
yes	2	HARD	9000	6	6999	20000	7394	130	RTS	2.29E+01
yes	1	HARD	17000	5	6875	20000	14341	130	RTS	3.87E+01
yes	5	HARD	18000	2	213	40000	14521	130	RTS	1.63E+03
yes	4	HARD	34000	1	142	40000	14735	130	RTS	3.63E+03

```
List of all protected objects:
```

PO#	Pr	Protected Type	PO User List
...			
4	8	RESOURCE	T#2, T#1,
5	7	SYNCHRONISATION	T#2, T#1,

```
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; and
- 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).

THREAD	TIME				ACHIEVED DEADLINE	
	CPU		RTS		min	max
	SUM	%	SUM	%		
CONSUMER.THREAD	2019928	33.70%	42861	0.72%	2930	9718
PRODUCER.THREAD	2005555	33.46%	88392	1.47%	1732	1811
...						

```
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 scenarii 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).

TIME	EVENT	THREAD	PO	PRIO
6865	ENTER_PROTECTED	PRODUCER.THREAD	CONSUMER.OBCS	6
6888	LEAVE_PROTECTED	PRODUCER.THREAD	CONSUMER.OBCS	7
6888	LOWER_BARRIER	PRODUCER.THREAD	CONSUMER.OBCS	7
6888	ENTER_PROTECTED	PRODUCER.THREAD	STORE.OBCS	6
6925	LEAVE_PROTECTED	PRODUCER.THREAD	STORE.OBCS	11
6925	ENTER_PROTECTED	PRODUCER.THREAD	PRINT_TOOL.OBCS	6
6948	LEAVE_PROTECTED	PRODUCER.THREAD	PRINT_TOOL.OBCS	10
6948	LOWER_BARRIER	PRODUCER.THREAD	PRINT_TOOL.OBCS	10
6948	HOLD_THREAD	PRODUCER.THREAD		6
7008	LEAVE_PROTECTED	PRINT_TOOL.THREAD	PRINT_TOOL.OBCS	10
7011	ENTER_PROTECTED	PRINT_TOOL.THREAD	STORE.OBCS	9
7047	LEAVE_PROTECTED	PRINT_TOOL.THREAD	STORE.OBCS	11
7051	ENTER_PROTECTED	PRINT_TOOL.THREAD	STORE.OBCS	9
7087	LEAVE_PROTECTED	PRINT_TOOL.THREAD	STORE.OBCS	11
7093	ENTER_PROTECTED	PRINT_TOOL.THREAD	PRINT_TOOL.OBCS	9
7109	SPORADIC_WAIT	PRINT_TOOL.THREAD		10
7109	SELECT_AND_CONTEXT_S	PRODUCER.THREAD		6
7148	RESUME_THREAD	PRODUCER.THREAD		6
7187	CYCLIC_SUSPEND	PRODUCER.THREAD		6
7187	SELECT_AND_CONTEXT_S	CONSUMER.THREAD		5
7226	ENTER_PROTECTED	CONSUMER.THREAD	CONSUMER.OBCS	5
7251	LEAVE_PROTECTED	CONSUMER.THREAD	CONSUMER.OBCS	7
7251	ENTER_PROTECTED	CONSUMER.THREAD	BUFFER.OBCS	5
7281	LEAVE_PROTECTED	CONSUMER.THREAD	BUFFER.OBCS	8
7281	ENTER_PROTECTED	CONSUMER.THREAD	STORE.OBCS	5
7318	LEAVE_PROTECTED	CONSUMER.THREAD	STORE.OBCS	11

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.)

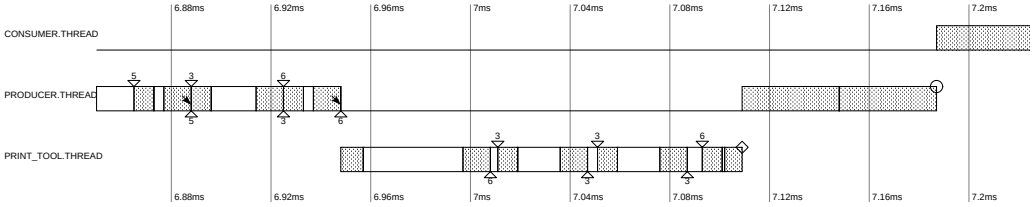


Figure 5.14: Scheduler Simulator Output (Gantt).

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 technical report. ([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 technical report 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**.