Designing Hard Real-time Systems

A. Burns and A.J. Wellings

Real-time and Distributed Systems Research Group, Department of Computer Science University of York, Heslington, York, Y01 5DD, UK

ABSTRACT

This paper presents a systems life cycle and a structured design method which are tailored towards the construction of real-time systems in general, and hard real-time systems in particular. The standard systems life cycle is modified to take into account the expression and satisfaction of non-functional requirements. The HOOD design method is extended to support abstractions which explicitly cater for the characteristics and properties of hard real-time systems. The new method is called HRT-HOOD (Hard Real-time HOOD).

1. Introduction

The most important stage in the development of any real-time system is the generation of a consistent design that satisfies an authoritative specification of requirements. Where real-time systems differ from the traditional data processing systems is that they are constrained by non-functional requirements (e.g. dependability and timing). Typically the standard design methodologies do not cater well for expressing these types of constraints.

The objective of this paper is to present a hard real-time systems life cycle, and a structured design method which is tailored towards the construction of real-time systems in general, and hard real-time systems in particular. Rather than developing a new method from scratch, the HOOD method is used as a baseline. HOOD (Hierarchical Object Oriented Design) has been chosen because it has a systematic mapping of the detailed design into Ada. The new method is called HRT-HOOD (Hard Real-time HOOD).

Section 2 of this paper presents the modified systems life cycle, and section 3 then discusses HRT-HOOD which has been designed to support the new life cycle. HRT-HOOD has also been developed to be compatible with the Ada 9X real-time tasking model[5]. Finally in section 4 we give our conclusions.

2. Overview of the Design Process

It is increasingly recognised that the role and importance of non-functional requirements in the development of complex critical applications has hitherto been inadequately appreciated[2]. Specifically, it has been common practice for system developers, and the methods they use, to concentrate primarily on functionality and to consider non-functional requirements comparatively late in the development process. We believe that this approach fails to produce safety critical systems. For example, often timing requirements are viewed simply in terms of the performance of the completed system. Failure to meet the required performance often results in ad hoc changes to the system.

Non-functional requirements include dependability (e.g. reliability, availability, safety and security), timeliness (e.g. responsiveness, orderliness, freshness, temporal predictability and temporal controllability), and dynamic change management (i.e. incorporating evolutionary changes into a non-stop system). These requirements, and the constraints imposed by the execution environment, need to be taken into account throughout the system development life cycle. During development an early binding of software function to hardware component is required so that the analysis of timing and reliability properties of a still unrefined design can be carried out[6].

Most traditional software development methods incorporate a life cycle model in which the

problems will only be recognised during testing, or worse after deployment.

2.1. Addressing Hard Real-time Issues

We believe that if design methods are to address hard real-time issues adequately they must support:

- the explicit recognition of the types of activities/objects that are found in hard real-time systems (i.e. cyclic and sporadic activities);
- the explicit definition of the application timing requirements for each object;
- the definition of the relative importance of each object to the successful functioning of the application;
- the explicit definition and use of resource control objects;
- decomposition to a software architecture that is amenable to schedulability and timing analysis.

In addition, design methods must allow the schedulability analysis to influence the design as early as possible in the overall design process, and restrict the use of the implementation language so that worst case execution time analysis can be carried out.

2.2. The Hard Real-time Life Cycle

Our approach is to split the architectural design into two phases[2]:

- logical architecture;
- physical architecture.

The logical architecture embodies commitments which can be made independently of the constraints imposed by the execution environment, and is primarily aimed at satisfying the functional requirements. The physical architecture takes these and other constraints into account, and embraces the non-functional requirements. The physical architecture forms the basis for asserting that the application's non-functional requirements will be met once the detailed design and implementation have taken place. It should be possible, for example, to prove that if all objects are built to their worst case timing and reliability constraints then the system itself will meet its safety requirements. In general, the physical architecture allows arguments to be developed that assess compliance with all the application's requirements.

In this paper we are primarily concerned with hard real-time systems, therefore the Physical Architecture Design is focused on timing requirements and the necessary schedulability analysis that will ensure (guarantee) that the system once built will function correctly in both the value and time domains. To undertake this analysis it will be necessary to estimate the execution time of the proposed code, and to have available the time dependent behaviour of the target processor and other aspects of the execution environment.

Once the Architectural Design Phases are complete, the detailed design can begin in earnest and the code for the application produced. When this has been achieved, the execution profile of the code must be measured to ensure that the estimated worst case execution times are indeed accurate. If they are not (which will usually be the case for a new application), the physical architecture design phase is revisited with the up-to-date information. If the system is not feasible then either the detailed design must be revised (if there are small deviations), or the designer must return to the logical architecture design phase (if serious problems exist). When the code measurement indicate that all is well, testing of the application proceeds. This should involve actual timing of code.

2.3. Logical Architecture Design

There are two aspects of any design method which facilitate the logical architecture design of hard realtime systems. Firstly, explicit supports must be given to the abstractions that are typically required by hard real-time system designers. Secondly, the logical architecture should be constrained so that it can be analysed during the Physical Architecture Design phase. These aspects are now discussed. of terminal objects with all their interactions fully defined. It is assumed that some form of functional decomposition process has lead to the definition of these objects.

The terminal objects are characterised as:

- CYCLIC,
- SPORADIC,
- PROTECTED, or
- PASSIVE.

CYCLIC and SPORADIC activities are common in real-time systems; each should contain a single *thread* that is scheduled at run-time. The priority of the thread will be set during the schedulability analysis of the physical architecture phase. PROTECTED objects control access to data that is accessed by more than one thread (i.e. CYCLIC or SPORADIC object); in particular they provide mutual exclusion. On a single processor system this will be achieved by having a ceiling priority defined for each PROTECTED object that is at least the maximum of the threads that use it. When a thread accesses a PROTECTED object it will run with this ceiling priority and hence have mutually exclusive access over the data hidden within the PROTECTED object. PROTECTED objects are also similar to monitors in that they can block a caller if the conditions are not correct for it to continue. This will be used to hold a SPORADIC object until the release event has occurred. The final important object type is PASSIVE which is used for an object that is either used by only one other object or can be used concurrently without error.

All the above four objects types are admissible as terminal objects in a hard real-time system. It is however possible that a real-time system may have a subsystem that is not real-time. The objects in such a subsystem are either PASSIVE or ACTIVE. ACTIVE object types may also be used during decomposition of the main system but must be transformed into one of the above types before reaching the terminal level.

With these types of terminal objects the common paradigms used in hard real-time systems can be supported:

- Periodic activities represented by CYCLIC objects
- Sporadic activities represented by SPORADIC objects
- Precedence constrained activities Precedence constrained activities involves a series of computations through terminal objects. They are likely to occur in a design which must reflect *transaction* deadlines.

The Logical Architecture Design Process may commence with the production of ACTIVE and PASSIVE objects, and by a process of decomposition will lead to the production of terminal objects of the appropriate character. For example, a required cyclic transaction from input to output may be first represented as a single ACTIVE object but may then be realised as a CYCLIC object followed by a series of SPORADIC objects linked by PROTECTED objects.

2.3.2. Constraining the Design for Analysis

In order to analyse the full design, certain constraints are required. These are mainly concerned with the allowed communication/synchronisation between objects. They are:

- (a) CYCLIC and SPORADIC objects may not call arbitrary blocking operations in other CYCLIC or SPORADIC objects.
- (b) CYCLIC and SPORADIC objects may call asynchronous transfer of control operations in other CYCLIC or SPORADIC objects.
- (c) PROTECTED objects may not call blocking operations in any other object.

Points (a) and (b) require that CYCLIC and SPORADIC objects are only allowed to communicate via fully asynchronous message passing or PROTECTED objects. Fully asynchronous implies that neither the

the logical design process.

2.4. Physical Architecture Design

For the purpose of this paper, the focus of the Physical Architecture Design is the timing requirements. The design process must support the definition of a physical architecture by:

- 1) allowing timing attributes to be associated with objects,
- 2) providing a framework from within which a schedulability analysis of the terminal objects can be undertaken, and
- 3) providing the abstractions with which the designer can express the handling of timing errors.

The physical design must of course be feasible within the context of the execution environment. This is guaranteed by the schedulability analysis. Issues of reliability must also be addressed during this phase. As we are not concerned with distribution here, many of the options for increased availability are not appropriate. There may be independent stand-by systems but they are outside the scope of the system actually being designed.

2.4.1. Object Attributes

All terminal objects have associated real-time attributes. Many attributes are associated with mapping the timing requirements on to the logical design (e.g., deadline, importance). These must be set before the schedulability analysis can be performed. Other attributes (such as priority etc.) can only be set during this analysis.

Each CYCLIC and SPORADIC object has a number of temporal attributes defined, for example:

- The period of execution for each CYCLIC object.
- The minimum arrival interval for each SPORADIC object.
- Deadlines for all sporadic and cyclic activities.

Two forms of deadline are identified. One is applied directly to a sporadic or cyclic activity. The other is applied to a precedence constrained activity (transaction); here there is a deadline on the whole activity and hence only the last activity has a true deadline. The deadlines for the other activities must be derived so that the complete transaction satisfies its timing requirements (in all cases).

To undertake the schedulability analysis the worst case execution time for each thread and all operations (in all objects) must be known. After the logical design phase these can be estimated (taking into account the execution environment constraints) and appropriate attributes assigned. Clearly, the better the estimates the more accurate the schedulability analysis. Good estimates come from component reuse or from arguments of comparison (with existing components on other projects). During detailed design and coding, and through the direct use of measurement during testing, better estimates will become available which will require the schedulability analysis to be redone.

2.4.2. Schedulability Analysis

Schedulability analysis is an integral part of the development of a physical architecture. The proposed design must be feasible; that is all deadlines must be guaranteed for all foreseeable circumstances. To do this requires knowledge of the processor speed, memory speed and memory capacity, plus kernel timings e.g. context switch. The timing behaviour of other hardware devices may also need to be known. If we assume the execution environment supports preemptive priority based dispatching of threads then the scheduling analysis is concerned with defining static priorities for the threads embodied within cyclic and sporadic activities. A number of formulae are available in the literature. If the system consists of mainly CYCLIC objects and they have period equal to deadline then the rate monotonic theories can be used[7]. A more flexible object structure is supported by deadline monotonic theory[1].

In summary, the schedulability analysis will add the following annotation to the terminal objects of the physical architecture:

• Centing priority of PROTECTED objects.

In addition, budget times that have been derived from the worst case execution times will be available.

2.4.3. Handling Timing Error

The schedulability analysis described above can only be effective if the estimations/measurements of worst case execution time is accurate. Within the timing domain two strategies can be identified for limiting the effects of a fault in a software component:

- Do not allow an object to use more computation time than it requested.
- Do not allow an object to execute beyond its deadline.

One would expect a design method to allow a designer to specify the actions to be taken if objects overrun their allocated execution time or miss their deadlines. In both of these cases the object could be informed (via an exception) that a timing fault will occur so that it can respond to the error (within the original time frame, be it budget or deadline). There is an obligation on the execution environment to undertake the necessary time measurements and to support a means of informing an object that a fault has occurred. There is also an obligation on the coding language for language primitives that will allow recovery to be programmed.

3. Hard Real-time HOOD

In this section we present the Hard Real-time HOOD (HRT-HOOD) design method which has been developed to support the ideas introduced above. HRT-HOOD is an extension of HOOD[4], and directly represents the abstractions considered in Section 2.

HRT-HOOD is based on the premise that:

- 1) it should be possible to express explicitly the characteristics and properties of hard real-time systems in the design method; and
- 2) it should be possible to distinguish at the design level the difference between an object which has an active thread of control and an object which is used to synchronise and pass data between active objects.

Consequently, HRT-HOOD has object types which represents the abstractions presented in Section 2.3.1. A hard real-time program will contain at the terminal level only CYCLIC, SPORADIC, PROTECTED and PASSIVE objects. ACTIVE objects, because they cannot be fully analysed, will only be allowed for background activity. There will, of course, be used during system decomposition to represent more abstract entities.

HRT-HOOD distinguishes between the synchronisation required to execute the operations of an object and any internal independent concurrent activity within the object. The synchronisation agent of an object is called the Object Control Structure (OBCS) (in Ada 9X this will normally be a protected record). The concurrent activity within the object is called the object's THREAD. The thread executes independently of the operations, but when it executes operations the order of the executions is controlled by the OBCS.

HRT-HOOD also has the concept of object attributes — which allow the expression of real-time attributes such as deadline, worst-case execution time etc.

In the following sections the details of the HRT object types and object attributes are discussed.

3.1.1. PASSIVE

PASSIVE objects which have no control over when invocations of their operations are executed, and do not spontaneously invoke operations in other objects Whenever an operation on a PASSIVE object is invoked, control is immediately transferred to that operation. Each operation contains only sequential code which does not synchronise with any other object (i.e. it does not block). A PASSIVE object has no OBCS and no THREAD.

3.1.2. ACTIVE

ACTIVE objects which may control when invocations of their operations are executed, and may spontaneously invoke operations in other objects. An ACTIVE object has may have an OBCS and one or more THREADs.

The operation available on ACTIVE objects are very similar to those provided by HOOD[4].

3.1.3. PROTECTED

PROTECTED objects are used to control access to resources which are used by hard real-time objects. PROTECTED objects may control when invocations of their operations are executed, and do not spontaneously invoke operations in other objects; in general PROTECTED objects may *not* have arbitrary synchronisation constraints and must be analysable for their blocking times. The intention is that their use should constrain the design so that the run-time blocking for resources can be bounded (for example by using priority inheritance[8], or some other limited blocking protocol such as the immediate priority ceiling inheritance associated with the Ada 9X[5] protected records).

PROTECTED objects are objects which do not necessarily require independent threads of control. A PROTECTED object does have an OBCS but this is a monitor-like construct: operations are executed under mutually exclusive, and functional activation constraints may be placed on when operations can be invoked. For example, a bounded buffer might be implemented as a PROTECTED object.

A single type of constrained operations is available on PROTECTED objects. It is:

• Protected synchronous execution request (PSER).

A constrained operation can only execute if no other constrained operation on the PROTECTED object is executing; it has mutually exclusive access to the object.

A PSER type of request can have a functional activation constraint which imposes any required synchronisation. However, for hard real-time systems these should not be used unless the time that a calling object may be blocked can be bounded.

PROTECTED objects may also have non-constrained operations, which are executed in the same manner as PASSIVE operations.

3.1.4. CYCLIC

CYCLIC objects represent periodic activities, they may spontaneously invoke operations in other objects, but the only operations they have are requests which demand immediate attentions (they represent asynchronous transfer of control requests). They are active objects in the sense that they have their own independent threads of control. However, these threads (once started) execute irrespective of whether there are any outstanding requests for their objects' operations. Furthermore, they do not wait for any of their objects' operations at any time during their execution. Indeed, in many cases CYCLIC objects will not have any operations.

In general CYCLIC objects will communicate and synchronise with other hard real-time threads by calling operations in PROTECTED objects. However, it is recognised that some constrained operation may be defined by a CYCLIC object because:

• other objects may need to signal a mode change to the cyclic object — this could be achieved by having CYCLIC objects poll a "mode change notifier" PROTECTED object, but this is inefficient if the response time required from the CYCLIC object is short (if mode changes can occur only at well

• other objects may need to signal error conditions to the cyclic object — this could again be achieved by having CYCLIC objects poll an error notifier PROTECTED object but this is again inefficient when the response time required from the CYCLIC object is short

Several types of constrained operations are therefore available on CYCLIC objects (each may have functional activation constraints). All of these, when open, require an immediate response from the CYCLIC's thread. The OBCS of a CYCLIC object interacts with the thread to force an asynchronous transfer of control. Available operations include an:

• Asynchronous, asynchronous transfer of control request (ASATC). This does not block the calling object but demands that the CYCLIC object responds "immediately". The request will result in an asynchronous transfer of control in the CYCLIC object's thread.

All CYCLIC objects have a thread, whereas only those with operations have an OBCS.

3.1.5. SPORADIC

SPORADIC objects represent sporadic activities; SPORADIC objects may spontaneously invoke operations in other objects; each sporadic has a single operation which is called to invoke the thread, and one or more operations which are requests which demand immediate attentions (they represent asynchronous transfer of control requests).

SPORADIC objects are active objects in the sense that they have their own independent threads of control. Each SPORADIC object has a single constrained operation which is called to invoke the execution of the thread. The operation is of the type which does not block the caller (ASER); it may be called by an interrupt. The operation which invokes the sporadic has a defined minimum arrival interval, or a maximum arrival rate.

A SPORADIC object may have other constrained operations but these are requests which wish to affect immediately the SPORADIC to indicate a result of a mode change or an error condition. As with CYCLIC objects, ASATC operations are possible. A SPORADIC object which receives a asynchronous transfer of control request will immediately abandon it current computation.

SPORADIC objects may also have non-constrained operations, which are executed in the same manner as PASSIVE operations.

3.2. Real-time Object Attributes

HOOD does not explicitly support the expression of many of the constraints necessary to engineer realtime systems. In the object description language there is a field in which the designer can express "implementation and synchronisation" constraints. Rather than use this to express an object's real-time attributes, a separate REAL-TIME ATTRIBUTES field has been added. These attributes are filled in by the designer normally at the TERMINAL object level. It is anticipated that many of the values of the attributes will be computed by support tools.

The following attributes are required:

• DEADLINE

Each CYCLIC and SPORADIC object must have a defined deadline for the execution of its thread.

• OPERATION_BUDGET

Each externally visible operation of an object must have a budget execution time defined.

An operation which overruns its budgeted time is terminated. Each externally visibly operation of an object, therefore, must have an internal operation which is to be called if the operation's budget execution time is violated.

• OPERATION_WCET

Each externally visibly operation of an object must have a worst case execution time defined. The worst case execution time for the external operation is the operation's budget time plus the budget time of the internal error handling operation.

of its thread of execution. An overrun of the budgeted time results in the activity being undertaken by the thread being terminated.

Each CYCLIC and SPORADIC object must have an internal operation which is to be called if its thread's budget execution time is violated.

• THREAD_WCET

Each CYCLIC and SPORADIC object must have a worst case execution time defined for its thread of execution. The worst case execution time for the thread is the thread's budget time plus the budget time of the internal error handling operation.

• PERIOD

Each CYCLIC object must have a defined period of execution.

• MINIMUM_ARRIVAL_TIME or MAXIMUM_ARRIVAL_FREQUENCY

Each SPORADIC object must have either a defined minimum arrival time for requests for its execution, or a maximum arrival frequency of request.

• PRECEDENCE CONSTRAINTS

A THREAD may have precedence constraints associated with its execution.

• PRIORITY

Each CYCLIC and SPORADIC object must have a defined priority for its thread. This priority is defined according to the scheduling theory being used (we are currently using deadline monotonic scheduling theory[1]).

• CEILING_PRIORITY

Each PROTECTED, CYCLIC or SPORADIC object must have a defined ceiling priority. This priority is no lower than the maximum priority of all the threads that can call the object's constrained operations.

This list may be extended - for example some HRT approaches may require minimum/average execution times, utility functions etc.

4. Conclusions

In this paper we have presented a hard real-time life cycle and illustrated how the structured design method of HOOD can be modified to make it more appropriate for hard real-time system design. The HRT-HOOD method has been influenced by the tasking model of Ada 9X, and programs designed by the method have a systematic mapping to the language (see Burns and Wellings[3] for details of the mapping).

Acknowledgement

The authors would like to thank Eric Fyfe and Chris Bailey of British Aerospace, Space Systems for their comments on the material presented in this paper. We would also like to thank Paco Gomez Molinero and Fernando Gonzalez-Barcia of the European Space Agency (ESTEC).

The work presented in this paper has been supported, in part, by the European Space Agency (ESTEC Contract 9198/90/NL/SF).

References

- 1. N.C. Audsley, A. Burns, M.F. Richardson and A.J. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach", *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, USA (15-17 May 1991).
- A. Burns and A. M. Lister, "A Framework for Building Dependable Systems", *Computer Journal* 34(2), pp. 173-181 (1991).

- (1992).
- 4. A. Burns and A.J. Wellings, "HRT-HOOD: A Design Method for Hard Real-time Ada", *Real-Time Systems* 6(1), pp. 73-114, Also appears as YCS 199, Department of Computer Science, University of York (1994).
- 5. Intermetrics, "Draft Ada 9X Mapping Document, Volume II, Mapping Specification", Ada 9X Project Report (August 1991).
- 6. H. Kopetz, "Design Principles for Fault Tolerant Real Time Systems", MARS Report, Institut für Technische Informatik, 8/85/2 (1985).
- 7. J.P. Lehoczky, L. Sha and V. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", Tech Report, Department of Statistics, Carnegie-Mellon (1987).
- 8. L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronisation", *IEEE Transactions on Computers* **39**(9), pp. 1175-1185 (September 1990).