‘Engineering’ the software in systems

by Derrick Morris, Peter Green and Richard Barker

The paper describes a method and notation for designing the software in embedded and other reactive systems. The design method is described in the context of a structured life-cycle, which recognises both functional and non-functional requirements, and it is illustrated by application to a substantial example. Mainly, for reasons of reuse and maintenance, an object-oriented solution is an implementation goal. The method focuses on producing software fit for its intended purpose in terms of user functionality, while being concerned with other aspects of product quality. It also seeks to efficiently utilise the varied skills and experience in a project team, and assist the team in distributing and meeting responsibilities. Commercially available CASE tools are adapted to support the method.

1 Introduction

Today, software developers have to face not only the inherent difficulties of producing quality software [1], but also the plethora of methods, notations and tools currently on offer. This is against a background of ever-increasing complexity in the systems that are required, and pressure for better reliability and ‘time to market’ achievements. The demands are exacerbated/increased by the advances that have been made in the capabilities of micro-electronic technology, and by the apparent success of hardware colleagues in improving their development process through the use of hardware specification languages, simulation, performance modelling and synthesis [2]. Thus, there is no escaping the need to improve the software engineering process, particularly in the way that it is applied to embedded systems.

Many methods, notations and tools are available, and difficult questions must be faced, such as which is the best method, how is it to be applied in practise, can the transitional costs be accommodated, and will the result be worth the effort and cost? We have had to face these questions as part of both teaching and research, particularly with respect to the software component of embedded systems. Although the term ‘embedded systems’ conveniently identifies the market niche, the most significant features of such systems from the developer’s perspective are that they are usually ‘reactive’ systems [3], they are developed by teams of mixed ability, skills and experience, and their designers have to operate in a constrained design space.

The approach to embedded system development discussed in this paper results from the integration of a number of familiar concepts, notations and methods into a novel and coherent framework for analysis, design and implementation. By making extensive use of abstract models, it ensures that at each stage of development both functional and non-functional requirements are met, and it also provides a route into an implementation consistent with the model. Pragmatic goals include achieving a structure that eases maintenance and creates reusable components. Throughout the paper, the method is illustrated by reference to an example system, described below.

2 System product example

The product example (Fig. 1) is a special-purpose operating system (OS). Although not an embedded system, it is highly reactive and exhibits many of the same features and problems. Three other factors make it an appropriate choice for this work. First, there are few stated requirements, and these can be understood without lengthy description. Secondly, the complexity of an OS is significant, and finally, the system has a substantial number of predetermined constraints. Thus, there is a clear need for applying two of the main features of the method, i.e. the evolution of system specification through models, and the development of a design subject to strong non-functional requirements.

Another reason for choosing this example concerns the relationship between OSs and embedded systems. Ideally, software for embedded systems should be written for a bare machine, to provide flexibility to optimise response times, to minimise the size of the software, and to integrate with application-specific hardware. However, this is a difficult development route to follow, particularly as even mass-market products are now using sophisticated 32-bit microprocessors, and so real-time OSs are often used. An alternative under investigation is to create a library of (object-oriented) parts, from which a customised OS can be constructed. The chosen example is a first step in this direction, demonstrating that the notion and method make the use of a bare machine less formidable.

The specific example to be used to illustrate the method is a multi-user OS for a bus-based multi-processor system.
that provides user access from dumb terminals, with password protection. To reduce the example to a suitable size, it is assumed that it will be sufficient to run only precompiled application programs, such as word processors, expert systems etc., and so program development facilities need not be provided. Hence, a task shell may be permanently allocated to each terminal, providing facilities for log-in and for the selection of application subsystems. These assumptions do not alter the key features that give the example relevance.

The initial product proposal is as follows. The interface required for applications should be a library of functions for basic input/output and file access. Program swapping and timesharing are to be as fast and unobtrusive as possible. Type ahead is required on input, and sufficient output buffering must be provided to maintain full output rates despite time-slicing. Only programs that are ‘trusted’ will be used, and so the library functions may run in the user space rather than in protected OS space. Four ‘supervisor call’ functions are required: load and release application, suspend process (used when input is requested and there is none in the buffer, for example), and terminate, which terminates a session and causes a re-initialisation of the task shell.

The hardware is assumed to be a caged system, containing at least three single board computers (one with a memory management unit capable of supporting virtual memory), a shared memory board, a multi-channel serial communications board (providing interrupts) connected to dumb terminals, and an SCSI board (not providing interrupts) connected to a suitable disk. One processor manages the file system, while another controls input/output, providing buffering in both directions. The third processor time-slices applications. Communication between software running on different processors is through shared memory. It will be seen later that this rather constraining, but not unrealistic, specification strongly influences the way in which the design develops, even from an early stage in the life-cycle.

3 Methods of analysis and design

One of the most heavily researched areas of software engineering over recent years concerns analysis and design methods, and it is appropriate to review this before deciding on the methods to be applied to the development of embedded and reactive systems. Two major categories of method merit particularly close examination. These are structured methods [4–6] and object-oriented methods [7–12]. Many other methods have also been proposed [13, 14], but these are more specialised towards the development of business information systems. Additionally, there is the Mascot [15] method, relevant because of its origins in the defence industries. This has not been considered here because it is less widely accepted and has lost ground to HOOD [10]. Formal methods have also been widely researched, but currently their penetration into industrial practice, at a system level, is more sparse and usually exploratory, and so they are not considered further.

Another opinion that influenced our selection of potential methods was that, regardless of other attributes, a method would not be suitable for embedded system products unless it encouraged the use of abstract models of system behaviour, expressed in diagrammatic form. This is because such systems normally require a strong element of invention, the outcome of which is subject to review and agreement. From early in development, there is a need to produce and expose ideas. The popular view that a definitive requirement specification can be obtained through the interrogation of customers breaks down with products that create new markets, or significantly broaden the application of existing products through upgrading. In these cases, there is a need to demonstrate new system proposals (at least through models) at a very early stage in the product life-cycle, in order to reach agreement on their suitability as a basis for further investment, and graphical models facilitate this better than text.

There is also wide agreement [16] that visual representations of complex systems improve the capacity of engi-
neers to handle the complexity. However, it must be admitted that the requirement for a diagrammatic notation does not narrow the choice of method much, because most structured and object-oriented methods have this characteristic, although the features of some fulfill the underlying needs better than others.

3.1 Structured methods

The structured methods chosen for closer examination involve structured analysis (SA) followed by structured design (SD) [4] and specialised variants for real-time systems [5, 6] (RTSA). SA produces a behavioural model of a proposed system through functional decomposition and expresses it as a network of concurrent processes. After the SA model has been validated and approved, a SD is produced that models the implementation of the system as a hierarchy of sequentially executing modules. The need for the structural transformation from an approved behavioural model to a different implementation model, and the scope for error that this introduces, is a cause for serious concern [8]. Nevertheless, structured methods still have to be seriously considered for embedded systems because of the level of usage they have already achieved, partly through the availability of supporting CASE tools such as Teamwork [17]. However, it has become very clear that the weight of research opinion has turned away from structured methods and towards object-orientation, with some of leading proponents of structured methods [8] also moving in that direction.

3.2 Object-oriented methods

Arguments supporting object-oriented (OO) software development appear in many publications, but their main thrust is consistent. OO analysis is argued to be the most natural way to understand and model the problem domain, with real-world entities prompting the choice of objects on which the abstract system model can be based. Moving from analysis to design is also considered to be simpler when using OO techniques, as it involves only incremental refinement and development rather than the creation of structurally different models.

The object concept, central to OO approaches, may be briefly stated as follows. An object is a software module that presents a crisply defined interface, encapsulating data and associated functions (or methods) relating to an abstraction of an application-relevant entity. As with abstract data types [18], the principle is that declarative statements are used to extend the basic type set of the language to include new types whose data can only be manipulated through specified functions.

The challenge in OO analysis and design is how to discover objects, and structure their class definitions, in a way that represents the best grouping of data and function. Clearly, in an information system, the primary purpose of many objects is to encapsulate the data that represents the attributes of application domain entities, and the functions will mainly provide controlled access to these attributes. Suitable structures for many objects can therefore be discovered by entity analysis. However, in all systems there is a need for some objects that play a more active role, concerned, with implementing system strategies, for example, and in practise the actual proportions of the different kinds of objects vary according to the nature of the application. For example, in reactive (process-intensive) systems, the bias is towards objects that provide algorithmically complex functions but contain little data. Entity analysis is not an adequate basis for object design in such systems, and alternative methods are needed, such as those based on the allocation of functional responsibilities to objects or 'agents' [19].

Other proposals for dealing with reactive systems have considered the merger of the structured and OO approaches, by using SA to front end an OO result [20, 21]. Part of the motivation for this derives from a desire to retain the best elements of a well tried and proven method (i.e. SA), but there is also a feeling that, in some applications at least, analysis of the system's functional behaviour is a legitimate 'up-front' issue. However, some workers [22, 23] consider the two approaches to be fundamentally incompatible, asserting that considerations of functional behaviour, applied at an early stage, will seriously deform the embryonic structure of a system.

Fortunately, there does at least appear to be a strong consensus that, whatever the initial approach, an OO implementation is desirable. OO programming can be demonstrated to yield a highly workable form of modularity, resulting in improved reliability, maintainability and the production of reusable software 'parts'. The strong movement towards the OO language C++ is largely in recognition of these benefits. If only the change to OO Analysis and Design was as clear cut.

3.3 Making a choice

Obviously, for a given system, some methods of software development are more suitable than others. It is instructive therefore to view the full spectrum of application types, and the range of methods and tools on offer, as a two-dimensional application/development process space. When making a choice, it is necessary to be clear where you stand in this space and to be clear at which points the chosen method needs to perform best. It is easy to be swayed by arguments that are true in one region of the space and to overlook that they may not be true elsewhere.

Fig. 2 summarises our subjective views on the range of applicability of several methods that have been examined (apologies to the originator for any method that is misrepresented). Although, as mentioned earlier, objects can be discovered by entity analysis, or by considering functional requirements and assigning functional responsibility to objects, most published methods emphasise the former. Thus, we do not believe that they are ideal for use with reactive systems. It should be noted that the responsibility-driven approach [19], which is applicable to object discovery in reactive systems, is not itself a complete development method.

Pragmatic considerations, such as the ease with which a method may be applied as part of a product development process, are important factors in the choice of such methods. From this point of view, we have found cause for concern with all the OO methods examined. Many of the difficulties in using OO methods stem from the fact that the problems concerned with identifying an appropriate set
of system objects are usually overwhelmed by the difficulties of constructing class definitions for the objects. This is typically done by producing hierarchical class definitions based on inheritance, so that not only is every object in the system an instance of one of these classes, but many of the class definitions are themselves built up by inheriting features from other classes. The concepts involved in generalising class structures can be viewed as adding unnecessary complexity to the already formidable task of developing a complex reactive system.

It is reasonable to expect, therefore, that those with practical experience in structured methods would prefer its more straightforward functional approach and will shy away from a sudden change to OO methods, which requires them to leave behind all that is good in structured methods. However, it has to be admitted that even ‘modern’ structured methods [4–6] do not offer the same promise for reuse and maintainability as OO methods, as it has to be agreed that abstractions based on functions are more volatile than those based on objects [7].

Thus, there is an attraction in seeking to evolve a method of producing an OO design from the general structured methods approach, despite views that the two paradigms are incompatible [22]. These views appear to be the result of a conservative interpretation of SA semantics. For example, the notion that the purpose of a process is to transform data persists, and data flows are usually regarded as passive communications channels that do not trigger actions. Separate control mechanisms are used to start, stop and trigger processes, rather than using the idea of data-triggered processes. In addition, the potential to use data packets as a transport mechanism for commands has not been exploited; doing so enables processes to be regarded as command processors. We believe that, with a more liberal interpretation of SA semantics, it can be used to produce a suitable structure for object models of reactive systems. This being so, the final step of replacing process semantics by object semantics is simple compared with learning a totally new method. Others [20, 21] who have attempted to adapt SA have not taken the same liberties with its semantics.

These conclusions have prompted the development of the method described in this paper, which uses CASE tools intended for SA with additional conventions and modified semantic rules. The development of models proceeds via hierarchical decomposition, in a manner akin to SA. Models contain a context diagram which shows the system under development as a single object interacting with externals (e.g. devices, users, other systems). The system object is then decomposed into a set of interacting objects, which together provide the overall functionality of the system and define its architecture. Each object may then be incremental refined until the resulting set of objects are deemed to be primitive, at which point textual specifications of the objects are produced. Thus, the method leads to a detailed specification of the OO code to be written.

In order to ensure the large-scale applicability of the ideas embodied in the method, they are integrated into a comprehensive life-cycle, discussed in Section 5. The notation used by the method is described below.

4 Modelling notation

Diagrams intended for SA are adapted to the role of documenting the models used throughout the proposed life-cycle, specifically state transition diagrams (STDs) and data-flow diagrams (DFDs). STDs are utilised to clarify aspects of system behaviour that are illustrated by introducing a temporal partitioning of functionality. They represent the external behaviour of the system, observed by a user (Section 5.2), and also specify the behaviour of control objects, which are used to handle interrupts (Section 5.5). DFDs are more extensively used in two different non-standard ways discussed below.

4.1 Object interaction diagrams

The object hierarchy starts with a context diagram that shows the system to be developed as a single object (‘bubble’), surrounded by the external entities (rectangles) that communicate with the system (Fig. 6). In the object interaction diagram (OID) hierarchy developed from this, each diagram represents a decomposition of an object at a higher level. The lines that connect each bubble are carried forward to the next level to become the lines that enter and leave the diagram (Fig. 7).
Three distinctive kinds of arrows are used on OIDs, representing interactions, information flows and events. As the CASE tool only provides two kinds of lines, solid and dashed, interactions and information flows use solid lines and are distinguished by their labels as described below. Events use dashed lines.

Interactions represent message paths that carry request/response pairs, the arrow showing the direction of the request, the response being implicit. When used on the context diagram of a user model (Section 5), they represent the reactive aspects of required system behaviour. Within the system they represent messages sent from one object to another requesting the execution of one of the receiving object’s operations. Information flows are unidirectional and are semantically similar to data flows in SA. Their purpose is to represent the information flow between the objects in a system and its externals, and they should not exist between objects. Production and consumption of information by objects only take place when objects are active, which is normally when they are processing an interaction message or event signals that represent hardware-generated signals (interrupts) and software traps. The responsibility for handling interrupts may be passed down the object hierarchy, but eventually resides in a special kind of primitive object, a control object, represented by a dashed circle, and discussed in Section 5.5.

The hierarchical nature of the notation is intended to allow ‘large’ conceptual chunks of a system to appear as objects at the higher levels of the diagram hierarchy, and then be progressively refined into simpler objects on subordinate ‘child’ diagrams. There is also some advantage in placing a similar hierarchical structure on the connections. For example, when there are several closely related operations, it is usually better to show them as a single line on the high-level diagrams, instead of individually. In addition, when several interactions are necessary to complete a single logical transaction between objects, readability can be improved by postponing the presentation of this detail until later. Thus, the notation allows connections to be grouped into ‘bundles’, which are replaced at lower levels by their constituents.

4.2 Name and label conventions

All object symbols and connecting lines on OIDs are labelled with text to which the following conventions apply. For primitive or library objects (see below), the text contains two names separated by a colon; the first name specifies the object’s class, and the second names the instance itself. Non-primitive objects are not given class names because the method does not require class definitions for them, but these may be synthesised and named by a support tool. Repeated instances of a particular class must all have the same class name followed by a unique object name.

Labels on connecting lines are chosen to convey an impression of their purpose, and this is supported by conventions. For example, the name of an interaction, which specifies one of an object’s operations, is prefixed by the words GET, PUT or DO, depending on whether the operation obtains or delivers data, or is algorithmically complex. Where appropriate, interaction names are also followed by parameter names in parentheses, which in turn may be followed by \ and a result name(s). Labels on interactions do not need to be unique, as their position on a diagram uniquely identifies them. However, this fails when several instances of one class occur on an OID and another object communicates with more than one of them, leading to ambiguity on the latter object’s OID. Thus, the labels of the appropriate interactions are prefixed with the object name to which they connect on the parent diagram.

Interaction and information flow names are written in lower case, whereas bundle names are written in upper case and have a dictionary entry (in the data dictionary provided by the CASE tool) which specifies the labels of the components, to which the above rules apply. Further conventions regarding the use of bundles are omitted in the interests of brevity.

4.3 Class implementation diagrams (CIDs)

The hierarchical decomposition of objects continues until objects are reached that are considered simple enough to be treated as primitive. The class of each such object is then specified by a class implementation diagram. On these diagrams (Fig. 10) the ‘bubbles’ represent functions and data stores depict class state information. Functions may be public (e.g. DO.Suspend Task) or private, the latter supporting the implementation of the former (e.g. Get Next Task). Constructor functions (e.g. Scheduler) may also optionally be used. There will be a single interaction line entering the diagram for each of the class’s public functions, connecting to the appropriate function. Functions within a CID may call each other, with calls depicted by arrows between functions. If a constructor exists, it has the same name as the class and is automatically executed when instances objects of the class are created. Constructors may have parameters which are enclosed in parentheses. Objects that produce or consume information flows (not shown in Fig. 10) will have such flows leaving/entering the CID and linking to the relevant function.

The specification of each object’s interface, which is all that should normally concern users of the class, is given in text linked to the CID. As the classes are intended to be reusable, the interface names must be appropriate to the abstraction represented by the class. These names may not be appropriate to any particular application. Thus, there is a further piece of text associated with each object that defines the mapping of its names into the equivalent class names.

Object models are complete once the textual specifications of each CID function and data structure have been provided.

5 Proposed life-cycle

Before presenting the proposed embedded system software development life-cycle, it is useful to postulate a simple overall model of the project management structure in which it will operate. It is assumed that when a new product development begins, a senior engineer (the ‘Project Engineer’) will be appointed to lead the effort and will have overall technical responsibility for the project. They will shape the high-level structure of the product, based on systems engineering judgments that take...
5.1 Analysing product proposals

The first phase of the life-cycle (Fig. 3) is concerned with producing a set of structured requirements. Following the method of Esteal and Davies [24], structured requirements are produced that distinguish between functional requirements, non-functional requirements, design objectives and design decisions. The first of these set behavioural goals for the system, and the others constrain the design space in which its designers operate. Clear recognition and confirmation of these constraints can be as important in launching a successful product development as approving its functional definition. However, the opposite view is often presented, curiously enough for the same motives, i.e. to avoid (wrong) decisions being taken by the wrong person at the wrong time [4, 24]. The apparent contradiction arises because embedded system software is only one part of a larger product, and consideration of the system as a whole inevitably leads to restrictions on the software designer's options, whereas in a large information system project, an unconstrained logical model is needed to set the scene for all implementation decisions.

5.2 Constructing the user model

The next phase of the life-cycle involves the creation of a user model. As in SA [4], this is an abstracted view of the system, based on functional requirements only, and it provides an external view of how the system is to behave. Clearly embedded system software may affect end users indirectly through other system components whose behaviour has been decided elsewhere. Thus, fine judgments are necessary to achieve a model that will satisfy the dual requirements of providing a basis for agreement on system behaviour, from a purely user perspective, and defining the overall functionality goals to be met in the subsequent 'systems engineering' phase.

The user model is a formalised deliverable of the life-cycle, in recognition of the fact that in speculative or innovative developments the initial product proposal is likely to be incomplete and even flawed, and appropriate clarification and assumptions are necessary to achieve consistency and completeness. As with the non-functional requirements, interaction is required with those responsible for the initial product proposal and its expected market, in order to gain agreement. Both the notation and style of the model, and indeed other models used later in the life-cycle, are critical to the efficiency of these human interactions.

The user model for the example is documented by a system context diagram (Fig. 4), a behaviour diagram (Fig. 5), which is an STD, and supporting text. In effect, the user model structures and clarifies the functional requirements. For example, the communication lines on the context diagram provide a logical classification of the total interface, and the behaviour diagram provides temporal partitioning of the system functionality. In both cases, clarification and detail are conveyed by associated text. As indicated in Section 4.2, there is a textual definition for bundles that describes their constituent parts. Thus, the bundle Supervisor Calls would be defined to be SuspendTask, TerminateTask, LoadApplication and ReleaseApplication. The text associated with each state of the behaviour diagram (not included here) will define the required behaviour of the system while in that state.

5.3 Systems engineering phase

The phase that follows agreement on the user model/structured requirements is where the critical 'systems
Fig. 5 Users view of system behaviour

Fig. 6 Architecture context diagram
responsibility and would not appear on the context diagram. Instead the kernel would be shown as an external, connecting with the system through the relevant commands.

In effect, the purpose of the diagrams that follow the context diagram is to decompose the system into its Primary Objects and to establish their responsibilities, which begs the question 'how are the functional responsibilities decided?'. Although the following guidelines do not provide a complete answer, we have found them helpful.

- Invent objects to serve as 'agents' with clear functional responsibilities [19].
- Assign to separate objects functions that will have real or quasi-concurrency.
- Ensure that there is strong cohesion for functions assigned to an object.
- Minimise the spread of responsibility for implementing action(s) associated with a single system state (as defined in the user model).
- Hierarchically decompose complex objects, stopping at a level that leaves maximum flexibility for the design to develop according to best software engineering practise, but ensure that the level of decomposition is sufficient to support some worst case timing analysis relating to event response times and information throughput.
- Show whatever detail is necessary to allow the specification of strategies that the project engineer deems critical to meeting objectives.

Figs. 7, 8a and 8b show the primary objects of the OS, and thereby complete the graphical part of the architecture model. The distribution of responsibility between primary objects of the OS is now briefly considered.

Responsibility for input/output buffering strategies rests with the communication channels object, and those for disk management with the disk system object. For both objects, the project engineer would probably impose performance constraints and would have a choice relating to the rigour of the functional specification. For example, operations could be fully specified or they may be left open for the present, which means that the authority to decide the best interface is delegated to the implementors of the objects. Where a specification is given, it is in the text associated with the object and/or the communication lines that enter it.

The disk command buffers and I/O buffers objects are interesting because they will be used by software running in two separate processors that communicate through shared memory, and these objects encapsulate the protocol that controls the communication. In the actual implementation, two instances of each object will be compiled, one into the software of each separate computer, but in the model their full responsibility is best defined by using one object.

The most complex object on Fig. 7 is the user task domain, and this leads to its functional decomposition (Fig. 8a) being treated as part of the architecture.

After the functional responsibilities of the primary objects have been allocated and documented in text linked to the object symbols on the diagram, appropriate constraints are added. These are decided according to the project engineer's interpretation of the structured requirements for the whole system, with the goal that if every object satisfies its constraints, those of the total system will be satisfied. Usually the project engineer will have only previous experience to guide the allocation of constraints, and decisions may sometimes have to be revised. However, clear documentation of constraints, even though they may be reviewed on the basis of later feedback from the design team, provide the project engineer with control over system issues that are often left to chance.

This feature of the life-cycle does, however, require that the project engineer's experience is sufficient to justify the prerogative to constrain the design space in which other team members operate. Of course, in practise the architecture is likely to be the result of a collaborative effort of senior members of the team, perhaps guided by the results of design studies and experiments. For example, if the architecture model is particularly complex, it is likely that the project engineer would commission analysis of critical system issues and possibly experiments, in order to build confidence that all non-functional requirements can be met when the further decomposition of primary objects is delegated. Reviews of the architecture based on 'what-if' scenarios might also be used to establish that the functional requirements are met. The notation of the model is intended to support and control such investigations. When the studies are complete, the architecture will be 'signed off', in the expectation that it firmly lays down the terms of reference for the further development of the primary objects.

5.4 Developing the hierarchical object model

Fig. 3 shows that there are two further stages in the life-cycle, after an architecture is produced and before the detailed coding begins. In the first of these stages a full hierarchical object model is produced, by continuing the decomposition of primary objects down to objects deemed simple enough to be treated as primitive. Both functional requirements and constraints are passed down this hierarchical model, so that at each level the designer's job is a specified and constrained task. During this stage, the reuse of previously defined objects (from a library) may be considered and potentially reusable objects may be invented. Once the object hierarchy has been completed, all the parts (and sub-assemblies) needed to construct the system have been specified. At the lowest level of the hierarchy, class interaction diagrams (CIDs) are used to specify the behaviour and data content of the primitive objects, as indicated in Section 4.3.

Up to this stage, a designer experienced in structured methods should not have the feeling of being on unfamiliar ground, provided that the requirement for good abstraction and strong encapsulation of detail by objects can be achieved. However, questions arise relating to the dynamics of the model. At this stage of the life-cycle they are similar to those of a SA model; the model is one of potentially concurrent objects, and those with no messages to process or events requiring responses will be 'busy waiting'. Behaviour of the model can be appraised by considering only the functions contained on the CIDs of
the primitive objects, as the behaviour of the hierarchy is equivalent to that of 'flattened' network that results from substituting the OID for each non-primitive object in the hierarchy and the CID for each primitive object. Messages sent to functions are assumed to be automatically queued if necessary, and communication is synchronous, with a sending function always waiting for a response from the receiver. Obviously these dynamics are not exactly those of OO languages, and this is one issue that is dealt with later in the life-cycle when the hierarchical model is transformed into an implementation model.

Research yet to be published has shown that the dynamics of hierarchical models can be investigated through interpretive execution, provided that suitable textual definitions are created for the operations of primitive objects. Executable models can be used to validate behaviour, explore performance, and later on, provide a test harness for implemented objects.

5.5 Developing the implementation model

The concurrency of the hierarchical object model will be restricted in the implementation according to whether the system is multi-processor or multi-tasking in nature, or is interrupt-driven. Objects that execute in a single task of a single processor will process their messages on the thread of execution of the senders. For software intended to be independent of OS support, the number of processors and the use of interrupts determine the number of threads of execution to be considered. Thus, the actual concurrency level of the implementation must be shown on the model so that it may be critically examined to ensure that deadlocks cannot occur, and that the integrity of the state information contained in the objects cannot be compromised. Another effect of the changes made to the model at this stage is that the low-level interface to hardware becomes the responsibility of control objects, and none of the other objects need to be concerned with events and state changes in the processor. A corresponding approach can be developed to handle the task threads if a standard OS kernel is to be used in the system.

A control object is introduced into the OID hierarchy for each processor in the implementation. Other related changes are that events to be implemented as interrupts in a processor are re-routed through its control object. In a similar way, interactions between objects that imply a state change in the processor, for example between user level and 'supervisor mode', are also sent through the control object. Fig. 8b is a revision of Fig. 8a, showing the addition of a control object and application of the above transformations.

The semantics of a model with control objects are that at power up only the control objects become active. Each control object may be 'busy wait', in which case other objects can only become active as a result of interrupts, or it may activate another object by sending it a message, thereby passing the thread of execution. When an interrupt occurs, the main thread of execution is paused and another message from the control object (taking with it the new thread of execution) activates the object(s) responsible for servicing the interrupt. This thread eventually returns to the control object, through the normal function return mechanism, where it terminates, allowing the main thread to resume.

A second transformation of the object model during this phase results from a review of the class structure specified by the CIDs. Here the full complexity of the OO philosophy can be applied, and 'kind of class hierarchies can be developed to produce the required class definitions. However, the amount of work and the skill level required are substantial, and the main benefit is in situations where there is potential for reuse. We have usually stopped short of this level of generalisation-specialisation elegance.
Instead we normally produce the CID for each primitive object and then construct its class definition, making only sparse, ad hoc use of inheritance. We realise that this will be considered to be an object-based, rather than an OO, solution which in principle could be more elegant, and some of our current research is concerned with methods for specifying inheritance on CIDs.

5.6 Route to the source code

The issues involved in translating the implementation model into source code are now considered. In order to be specific, C++ is taken as the target language.
ding of application-specific names in code for the member functions.

An alternative solution is to include pointers to objects whose methods are used, within the state space of a calling object, although additional member functions are then required to initialise these pointers at run time. This approach has the advantage that it can also be applied to the creation of class definitions for the non-primitive objects throughout the hierarchy, although doing so can be tedious. However, the task is a deterministic one and as such can be automated, thereby synthesising a skeleton of all the class definitions for both primitive and non-primitive objects. These will contain declarations of pointers to the class types used, definitions of pointer initialisation functions, and in the case of classes representing non-primitive objects, instances of primitive objects or other non-primitive objects. The constructors of non-primitive classes will also require calls to the pointer initialisation functions.

6 OS model

Selected parts of the OS model have already been introduced that illustrate the notation and some of the stages of the method. In this section we aim to use the model to highlight the principal virtues of the method through a discussion of the parts already presented, and by developing additional elements of the OS model.

Figs. 4 and 5, which depict part of the user model, clearly show the external view of the system in a form that is simple and easy to understand, features which are significant if the user model is to form an appropriate vehicle for communication between customers and developers. The architecture context diagram shown in Fig. 6 is a development of Fig. 4, which indicates how the logical interactions between the system and its environment are to be realised. This two-stage procedure is important in that it enables the logic of the required interactions to be established before considering their implementation, achieving a valuable separation of concerns. Once complete, the architecture context diagram rigorously defines the system's interface to its environment. Fig. 6 also illustrates that, although the notation is simple, it is rich enough to describe the range of possible interactions between an embedded system and its hardware environment.

Hierarchical decomposition is illustrated by Figs. 7, 8a and 9, which demonstrate how a simple structure, conforming to good OO practice, can be obtained with the help of the guidelines presented in Section 5.3, particularly with respect to functional responsibility, concurrency, and the partitioning of states in the behaviour diagram (Fig. 5).

Comparing Figs 8a and b again shows how the method enables the logical structure of the system to be established, and how details of interactions with hardware can be introduced into the model and isolated from the system's objects through the use of a control object.

It is not our intention to include the rest of the OS model, as it is much too large to present in this paper. However, the OS kernel has been chosen for further decomposition, as it demonstrates a number of additional features of the method. The decomposition of this object is shown in Fig. 9. The main criterion for choosing this decomposition was to cleanly separate responsibilities associated with tangible entities. Thus, objects are responsible for managing the virtual memory, scheduling tasks and controlling the allocation of RAM. The interfaces of these objects are established by considering both the services each object provides to its clients and also the facilities that each object requires. Careful choice of the latter is just as important as the former, a point that some other methods do not recognise.

Consider, for example, the VM manager object in Fig. 9. Its main responsibilities are to initialise and reset the virtual memories, to load and release applications and service

![Fig. 9 OS kernel OID](image-url)
virtual memory interrupts. In addition, it has a method (DO.PeriodicCheck) through which it is periodically activated in case it needs to carry out timed actions. Responsibility for calling this method is assigned to the control object on Fig. 8b because it receives timer events. Another instance of anticipating a need is seen in Fig. 9, in the passing of information from the VM object to the task scheduler (PUT DiskWait) that relates to tasks held up waiting for disk transfers to complete. The task scheduler may choose to act on this information or not according to what seems to be the best strategy, but if the information was not passed, it would not have the choice. In a similar way, the VM object can choose from a wide range of strategies how best to deal with VM interrupts, the simplest being to treat them as faults, but this, of course, has implications on the definition of the virtual store facility.

We assert that this kind of generalisation of object interfaces that provide flexibility for determining implementation strategies inside objects can only be achieved in a design method strongly biased towards identification of functional responsibilities. It is most relevant if the long-term intention is to build up libraries of reusable objects and, in the case of some objects, to offer several interface-compatible implementations of varying internal complexity.

Fig. 10 illustrates the class structure of the primitive task scheduler object, showing how the operations of the class affect the state of instances and how the operations interact with each other. It is evident that, by providing such a detailed class specification, the skeleton structure of the class, and indeed the whole system, can be synthesised from the model.

7 Experience in applying the method

We have constructed several models for products, such as video cassette recorders and PABXs, in order to assess the method and notation. Evaluation has also taken place in an undergraduate software engineering laboratory. This is attended by second-year students and provides their first experience of complex software design. Previously, SA, SD and C were used in the laboratory, and the OO method described in this paper has provided a satisfactory alternative that provides experience in C++. The students work in teams of five or six and are initially given a loose requirements specification for an embedded system, from which they deduce a set of structured requirements. These are used to guide the construction of a hierarchical object model and CIDs, leading finally to an implementation in C++.

All the projects involve some level of concurrency and low-level device handling, and so provide a reasonably challenging test of the method's applicability. Project timescales are very tight (12 weeks from start to finish), providing little room for many iterations of the model before it is frozen for implementation.

Of the projects undertaken so far, all of the groups have been able to apply the method, with resulting implementations which varied from fully integrated working systems to those where the primary objects had been implemented and successfully tested but not integrated. Comparisons made with previous years in which SASD and C have been used show an improvement in program structure. There is now little global state information, the grouping of data and function demonstrates better cohesion, and system components are more loosely coupled.

The experience of the students in the design laboratory leads us to be optimistic about the applicability of the method, the feasibility of learning it, the cost of developing systems and the quality of the software that results. In addition, our own work convinces us that it can be scaled up to industrial-sized problems. The students' designs show how the method allows issues of low-level interfacing and concurrency to be handled in a simple and comprehensible manner, and enables constraints to be propagated through the design so that the developer can assess whether the resulting implementations actually meet the associated constraints.

Fig. 10 Task scheduler CID
8 Conclusions

In this paper, we set out to convey what we have learnt from our attempts to establish a process for engineering the software in reactive systems. A full life-cycle has been proposed, together with a notation supported by an improvised CASE tool. The graphical models facilitated by the notation are central to the operation of the life-cycle, and form the primary means by which non-functional considerations influence design decisions and ensure that the final product meets all its requirements. They also aid design validation. There is sufficient detail in the models for them to be executable, and this feature can be exploited for investigating design options, validating choice and finally testing an implementation.

Although we concede that many methods have a broad range of applicability, and indeed the method presented here may be applicable to systems that would not be classified as reactive, it is reactive systems, and in particular embedded systems, that we have tailored the approach towards. Such systems are of growing importance, and we believe that other methods, unlike the approach proposed here, have not fully recognised some of the important issues which seriously affect embedded system development, such as early treatment of non-functional requirements, analysis of functional responsibility and the implications of reusable libraries.

We chose to base the notation on a concept of objects that map into class definitions in C++. The main reason was to obtain the recognised benefits of OO methods: the clarity of understanding that comes from abstraction, the safety of implementation that comes from encapsulation, and the potential for reuse. From our experiences of both teaching and using the approach, we believe that it works well. Based on this, we are currently directing effort into extending the method to form a total systems approach to hardware/software codesign and the construction of tools to support it.

9 References


© IEE: 1995

The paper was first received on 8 September 1994 and in revised form on 24 July 1995.

The authors are with the Systems Engineering Group, Department of Computation, University of Manchester Institute of Science and Technology, PO Box 88, Manchester M60 1QD, UK.