An Integrated Approach to Engineering Computer Systems

Derrick Morris, D. Gareth Evans & Peter N. Green
Computer Systems Design Group, Department of Computation, UMIST, Manchester, UK

Abstract

This paper describes MOOSE a full-lifecycle, model-based approach to the engineering of computer systems. It describes how early lifecycle models that represent the logical behaviour and architecture of a system can be transformed into representations that allow implementation source for both hardware and software to be synthesised.

1. Introduction

This paper presents an integrated development process covering the complete development lifecycle for Computer System-Based Products. It provides methods and tools that apply mainly to the front end of the lifecycle, and which integrate with existing tools to produce implementations.

A broad range of products form the anticipated application area, including special purpose distributed systems and custom built embedded systems that control a wide range of products, such as domestic appliances, portable personal appliances, communication systems, vehicle control systems and industrial control systems. The main features of these systems that are relevant to the approach are that they are application specific, subject to stringent non-functional requirements, embedded as defined by Zave [1] and reactive as defined by Harel [2]. Their implementations usually involve both software and hardware, which may be either analogue or digital.

Even though many examples of products based on computer systems exist, the discipline for developing them is by no means mature or stable and their development presents a daunting and challenging task, sometimes leading to serious problems [3]. Although the design and implementation of hardware and software components is fairly well understood, relatively little progress has been made in development processes which consider the systems as a whole. The approach described delays the separate treatment of hardware and software until a stage when high level implementation source can be automatically synthesised from a complete system design. Techniques and tools have been developed to support a development paradigm that starts with a model that specifies the behavioural requirements of a complete product. This behavioural model is then made executable in order to support validation and provide an environment in which system wide issues can be investigated prior to making the technology commitments on which an implementation will be based. Since models play a central role in the development method, and since they are Object Oriented, the approach is called Model-based Object Oriented System Engineering (MOOSE).

Traditionally software has usually been in overall control of system functionality, whilst the performance, size and power requirements of a system have been largely determined by its hardware. However, with the advent of ASIC technology and high performance RISC processors, there are now important trade-offs between hardware and software implementations to be investigated. The MOOSE approach recognises this situation, and that some non-functional objectives may only be achievable if functional responsibility can be easily moved between software and hardware. It follows that products exhibiting different cost/performance trade-offs may be developed from the same model, and that designs are re-usable across products.

This paper presents the MOOSE development paradigm for application-specific computer systems, and the notation and the tool support on which it is based. A portion of a simple control system model is presented, and there is a discussion of a significant aspect of novelty in the paradigm, namely Transformational Codesign.

2. Computer system development methods

Recently, an awareness has grown that methods are needed to support the integrated development of the software and hardware in application-specific systems, and this has lead to growth of hardware/software codesign. Currently, most codesign approaches do not attempt to cover the entire development lifecycle of a product, and focus on a single stage, namely the partitioning of a system into hardware and software

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00 ©1996 IEEE

components that realise an efficient implementation, typically optimised for price/performance. Normally these approaches are targeted towards a fixed implementation environment that has a pre-defined operating system, processor and support hardware, e.g.[4, 5, 6].

Other methods for the engineering of computer-based products have been proposed that consider the complete process. These include the initial specification of the operating behaviour of the product and its transformation into a hybrid implementation that includes application-specific software and hardware, system software and processors. One such set of approaches seeks to integrate existing analysis and design methods and tools to provide a seamless approach to complete system development, e.g. [7]. These approach have the advantage of using well tried methods at the expense of introducing tool and method integration problems. Alternative paradigms attempt to devise homogenous methods and tools that provide an integrated route to system development, e.g. [2, 8]. These have the advantage of being focused directly on the problem area, but have the corresponding disadvantage of needing either to invent completely new methods or to provide far-reaching extensions to existing methods, both cases requiring the development of supporting tools. For the reasons outlined elsewhere [9] the research introduced in this paper falls into the latter category, and is based upon cospecification, codesign and cosimulation, followed by automatic synthesis of implementation source from validated models.

3. Features of the MOOSE paradigm

As shown in Figure 1 the MOOSE paradigm starts from a product idea, and the MOOSE-specific part ends when the development can be completed by applying standard techniques and tools to implementation source for the hardware and software parts of the product, automatically synthesised from a 'proven' model.

For software, C++ is the current implementation language and the 'standard tools' are the compilers and debuggers for that language. However, this choice of language is not fundamental to the approach and it can adapt to lower level languages such as 'C' and Assembler, or to alternative higher level languages. If necessary, multiprocessor implementations can be developed, and the automatic synthesis mechanism produces separate program source for each processor. The objects placed in different processors, hence different programs, interact through communication mechanisms that are either developed or selected in the final stages of model development.

For hardware, the current implementation language is VHDL and the 'standard tools' are those that provide simulation and automatic manufacture of Application Specific Integrated Circuits (ASICS) from a VHDL specification. Again this choice is not fundamental, and a route exists within MOOSE to enable systems to be implemented on PCBs using standard parts.

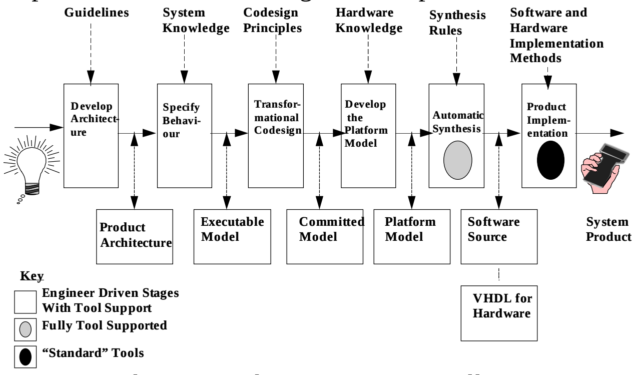

Figure 1: The MOOSE paradigm

Four phases of incremental development precede the synthesis of the implementation source. The first phase develops an architecture having three components, namely a Domain Model, an analysis of non-functional requirements and a Behavioural Model that captures the system's functional requirements. In the second phase the Behavioural Model is extended to become an Executable Model, so that the behaviour can be dynamically validated. During the third phase experiments are performed with the Executable Model and transformations are applied that capture the decisions committing objects to a software, firmware or hardware implementation. This phase is concluded by associating all three kinds of objects with specific processors. At this stage there is enough detail in the Committed Model to synthesise the complete C++ source for each processor in the implementation, with the exception of functions that interface to hardware objects or which facilitates interprocessor communication. In the fourth phase sufficient hardware detail is developed in a Platform Model to support the complete synthesis of the implementation source, for both the software and hardware.

The Platform Model is automatically created from the Committed Model. Its main purpose is to focus subsequent design effort on hardware detail such as register interfaces and bus and memory organisation. When completed, this model defines the interconnection of all the hardware components in the system, including application specific hardware, processors, memories, buses and other interconnect mechanisms. In addition it specifies the interfaces between application-specific software and hardware. There is enough information in this model for any style of hardware specification to be synthesised.

4. The modelling notation

MOOSE models have an Object Oriented structure, this is preferred to functional decomposition because of its potential for greater stability during design evolution, safer encapsulation of detail, and reuse. The Object Oriented notation used differs from those typically used to analyse software systems (such as OMT, etc.) in that it focuses on the objects that form the system rather than a classification of the application domain and their relationships. This approach is motivated by the desire to produce a system structure that provides a firm basis for the partitioning of the system and the subsequent synthesis of source code for the implementation.

An important aspect of a modelling notation is its ability to represent a system in manageable and readable portions with a well-organised structure. The hierarchical structure of Structured Analysis and HOOD perform well in this respect, and hence a similar structuring mechanism has been adopted for the MOOSE notation. Thus a complete model consists of a hierarchy of diagrams called Object Interaction Diagrams (OIDs), and the class definitions for these objects. At the higher levels of the hierarchy the objects on an OID represent subsystems whose decomposition into objects is given on subordinate diagrams. Decomposition stops at objects considered simple enough to be treated as *primitive*, and these are defined by class definitions.

The MOOSE notation provides a number of object to object communication types which represents abstractions of typical interaction mechanisms between computer system components. The semantics of these communication mechanisms have been defined so that executable models can be synthesised from the graphical model with comparatively little textual information being required. Moreover, the major mechanisms can be mapped directly into implementations in both hardware and software. Briefly the communication types provide the means for an object to call a function (method) of another object (*interaction*); make information available on a time continuous basis (*time continuous information flow*) and to broadcast *events*. In addition are there are *parameterised event* and *time discrete information flows* that are used to connect a system to its external interface. These, as we will discuss later, present an abstract view of the interface and are provided so that the commitment to a particular interface style can be postponed until the consequences of its implementation can be understood.

5. An example model

To demonstrate the principles of the modelling approach and illustrate the notation, the well known 'Mine Pump' control system has been chosen since its behaviour is simple but and it has just enough features to bring out the main principles of MOOSE.

Figure 1 indicates that if the water collecting at the bottom of a mine shaft rises above a certain limit the pump should be switched on, and when the water has been sufficiently reduced the pump should be switched off. The pump can also be turned on and off by a human operator. Any operator can control the pump when the water level is between the high and low sensors, and a specific operator designated the 'supervisor' has the authority to control the pump whatever the water level.

For safety reasons there are sensors monitoring methane (CH4) and carbon monoxide (CO) concentrations, and airflow; and an indication must be given of any critical values since evacuation will be required. Further, due to the risk of fire, the pump must not be operated when the atmosphere contains too much CH4. Finally all three sensors' values along with the pump status (i.e. on or off) should be periodically logged.

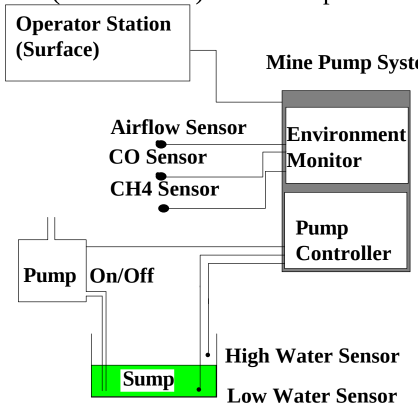

Figure 2 - Schematic of the mine pump system

6. The Behavioural Model for the mine pump

The *External View* is the top of the hierarchy of OIDs that make up the Behavioural Model. This is developed with two principles in mind. Firstly, it is constructed to clearly identify the scope of the system development, i.e. to precisely identify what must be developed. Secondly, it presents a statement of the behaviour of the system as viewed from its external interface. The *External View* may be augmented with a set of State Transition Diagrams so that the system’s behaviour is precisely defined.

The *External View* for the Mine Pump Control System is shown in Figure 3. Here the system is represented as a single object (a circle) that connects to a set of external objects (rectangles). The scope of the system can be clearly seen. For example, the use of the external object Gas Sensors implies that the design of these sensors is outside the scope of the Mine Pump System's development; if they were to be developed as part of the system, an external object 'Atmosphere' would have been chosen. The connections CO Concentration, etc. are defined electrical signals that relay the concentration and flow of gas to the system. The use of the external object Operator implies that the design and physical development of the user interface will be carried out as part of this project. If this were not the case, physical external objects such as a keypad would have been used. This interface also illustrates the principle of minimum commitment to a design decisions in the Behavioural Model. It shows that the operator can inform the system that the pump should be turned on via the *parameterised event* pump on. This event not only signals the operator's desire to start the pump but also carries as an associated parameter that indicates whether the operator has supervisor privileges. The precise interface mechanism will be developed in the *Transformational Codesign* phase, when the commitment to the physical interface, through, for example, a key lock, voice recognition or PIN number can be made in the context of the system's design. The postponement of interface decisions in this way allows the one Behavioural Model to serve as the starting point of many implementations in which different interface commitments may be made.

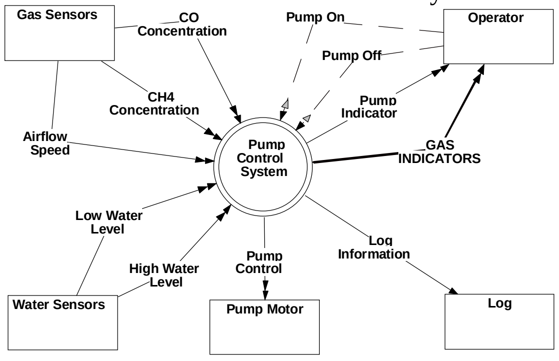

Figure 3: Mine pump model - External View

The Behavioural Model is developed from the External Model by decomposing the single system object into a set of communicating objects that are represented on an OID. The first OID for the Mine Pump Control System is shown in Figure 4. In this simple example all except the Gas object are primitive and the behaviour and purpose are described as text in an Object Specification (OSPEC). The Gas object is further decomposed into a set of objects represented on a lower level OID.

The Behavioural Model is developed with a number of principles in mind:

- Object oriented principles should be followed, therefore the objects should have neat encapsulation and crisp interfaces
- Commitment to implementation should be minimised. Thus the objects developed should be capable of being implemented in hardware or software.
- The model serves as an architecture from which the system is built.
- The model will express the maximum concurrency potential of the system; in the belief that it is easier to reduce the concurrency as one derives the implementation than it is to develop it.

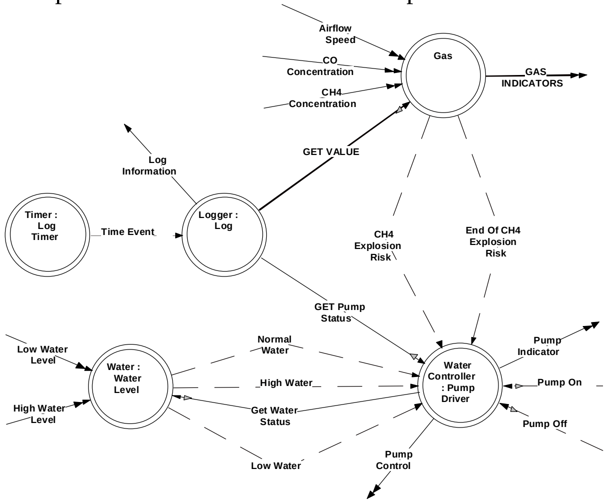

Figure 4: First level refinement of mine pump

The proposed system's dynamic behaviour is evaluated by the execution of a C++ program that is synthesised by tools from the model. This firstly requires that the Behavioural Model be developed into an Executable Model by specifying the class of each primitive objects is added. Class behaviour is specified through a combination of graphical notations and executable code.

This brief summary of the modelling notation has excluded a number of features, including those concerned with the dynamic creation of objects, the representation of repetitive patterns of objects and the development of class hierarchies, which introduce inheritance into the class definitions [9].

7. Transformational codesign

In this phase of the MOOSE paradigm implementation decisions are incorporated into the Executable Model. The goals are to make decisions that provide a 'best fit' with the non-functional requirements, and incorporate them by changes that are of a transformational nature, thereby safeguarding the 'proven' behaviour of the model. Since the model remains executable its logical behaviour and performance characteristics can be investigated as necessary.

Typically the steps to be taken in transforming an Executable Model into a Committed Model are:

- Design of the Interface Mechanisms (requiring the addition of objects and the transformation of connections)
- Commitment of Objects to Hard or Soft Implementations (requiring the modification of object inter-connections)
- Decisions on the Number of Processors (and the association of objects with them)
- Analysis of Threads of Execution

The design of the interface mechanisms is the first step because in general it requires the addition of new objects, and these are best added to the model before other decisions are taken. Thus each external connection to the model has to be examined in order to decide if its implementation requires special interface objects to be added. For example, in the case of the Mine Pump Control System we might decide to introduce a control panel in order to interface with the Operator. It is clear that the control panel has to deal with operator identification, pump operations, and to convey pump and gas. Figure 5, which is the final Committed Model, shows the addition of a User Console object.

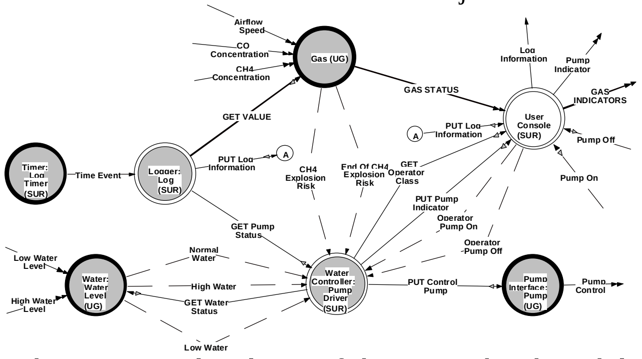

Figure 5: Top level OID of the Committed Model

The important features are that there is little disturbance to the model and the changes that are not purely additive involve only localised transformations. For example, the parameterised events are intercepted by the new object (User Console) behind which they become interactions. Thus the discovery of user identity becomes a responsibility of this object.

Next the commitment of objects to either software or hardware can be decided. The essential problem here is to satisfy the non-functional requirements. The MOOSE tools do not make the decisions but they assist in identifying the need for change and they provide, through the executable models, a means to investigate the consequences of various commitments. Finally they allow the implementation decisions to be captured in a manner that carries them through to synthesised implementations. Commitment is marked on an OID by shading the objects; shaded objects with a black border are hardware, with a white border are software and non-primitive objects that contain both hardware and software objects are left uncoloured - see Figure 5.

Following the hardware/software commitment, a review of inter-object connections is required and transformations to produce the preferred hardware to hardware and software to software connection style are made. That is, software mainly uses interactions and hardware mainly uses information flows and events. The hardware/software connections may involve interactions driven by software and events produced by hardware.

The next step is to decide on the processors that will be required, and several factors influence this choice. Obviously it determines the actual concurrency in the software, and this has to be matched with the need for intrinsic concurrency. Requirements for physical distribution of the system also affect the number of processors required. Clearly the choice of processor types combined with their number determines the performance available, which must be sufficient to meet peak demands and response times. Other non-functional requirements, such as power consumption may be the deciding issue on processor type, and this may impact the number of processors required to meet the performance requirements. The assignment of objects to processors is indicated by writing a logical processor name on the object. In this example (Figure 5) we have two, SURface and UnderGround. Experiments with the executable model provide a means for validating the performance to be expected.

After the processors have been chosen and objects have been assigned to them, the consequences for the threads of execution running through the soft objects have to be analysed. This analysis has to recognise that each processor provides a sequential (but interruptable) execution mechanism for the functions of the software objects assigned to it, whereas in the model prior to processor commitment each object was considered to be concurrent. Checks to be made concern, for example, deadlines and times to service events, reachability, integrity, and freedom from deadlock and livelock. Various transformations of the model may be needed if problems are highlighted by the above analysis. Again the Executable Model is a vehicle that supports the analysis.

8. Operations on the Platform Model

The Platform Model is automatically created from the

Committed Model. It omits all the Committed Model's software objects, since enough detail already exits to synthesise their C++ source, but introduces new software objects that provide: the interface between software and hardware; the communication between software in different processors; and the management of the threads of execution. These will all be developed as the Platform Model is refined. The hardware design is also completed. When the Platform Model is initially synthesised, with the exception of the processor, it contains only the application specific hardware objects, and new objects have to be added to provided the hardware glue in the form of buses, address decoders, memories, interrupt handlers, etc.

Finally, the definitions of the application specific hardware objects that exist in the Committed Model have to be translated into synthesisable VHDL. This process is supported by tools. In addition, appropriate tool support can allow the VHDL specification to be dynamically tested in the context of the model by using cosimulation techniques and thus its behaviour can be verified as being consistent with the emulation.

Thus the transformations to the Platform Model result in a model in which all components exist in a form whose implementations are synthesisable. The full synthesis of implementation from the Committed Model and the Platform Model results in the C++ source for each processor and a VHDL description for the all the system's hardware. The latter is, in most cases, directly synthesisable into an implementation.

9. Summary

Since this paper reports on ongoing work a status report is more appropriate than conclusions. The notation and basic approach to the development of application specific computer systems is stable and it is documented in a book [9], and a partial toolset is available.

The tools that support the MOOSE paradigm and notation presented take the form of a PC/Microsoft Windows-based workbench (MOOSEbench) catering for the early stages of the paradigm and providing model capture facilities, synthesis of an executable models, and an execution environment for running executable models. Further information and a trial version of the workbench is available on the World Wide Web at http://www.cl.co.umist.ac.uk/moose Work is currently underway to provide tool support for other parts of the paradigm. Ongoing tool development work is focused on the synthesis of implementation source code in C++ and VHDL. Other research is also underway which is primarily concerned with the introduction of automatic analysis to supplement or perhaps even replace some of the human effort that is still required to realise a successful product. The present level of automation can only be claimed to be improving product quality and human productivity.

The objectives of reuse of designs and implementations are met by the provision of libraries accessible through the workbench. In so far as automation of computer system design means reducing the work in designing computer systems, these libraries play a major role. The MOOSE libraries consist of reusable objects (i.e. their class definitions) and subsystems (i.e. object networks specified by OIDs). The MOOSE project is developing libraries catering for the use of implementation technologies developed within the ESPRIT Open Microprocessor Initiative (OMI) and selected application areas such as vision and multi-media.

A number of case studies, based on real systems products and in collaboration with industry, are currently in progress to assess the method.

11. Acknowledgements

This work has been partially funded by the EC (projects OMI/DE 6909 and OMI/MODES 20.592) and the EPSRC (project GR/J07631).

12. References

1. P. Zave, "The Operational Versus the Conventional Approach to Design", Comm. ACM., SE-9, No. 2, pp 104-118, 1984
2. D. Harel, "Biting the Silver Bullet: Towards a Brighter Future for System Development" IEEE Computer, Vol. 25, No. 1, pp. 8-20, 1992
3. W. W. Gibbs, "Software's Chronic Crisis", Scientific American, pp. 72-81, September 1994
4. R. Ernst, J. Henkel, J. and T. Benner, "Hardware-Software Co-synthesis for Microcontrollers" IEEE Design and Test of Computers, Vol. 10, No. 4, pp 64-75, 1993
5. S. Kumar, J. H. Aylor, B. W. Johnson and W. A. Wulf, "A Framework for Hardware/Software Codesign" IEEE Computer, Vol. 26, No. 12, pp 39-45, 1993
6. D. E. Thomas, J. K. Adams and H. Schmit, "A Model and Methodology for Hardware-Software Codesign" IEEE Design and Test of Computers, Vol. 10, No. 3, pp 6-15, 1993
7. K. Kronlof, Method Integration: Concepts and Case Studies. John Wiley & Sons, Chichester, 1993
8. N. S. Woo, A. E. Dunlop. and W. Woolf, "Codesign from Cospecification" IEEE Computer, Vol. 21, No. 1, pp 42-47, 1994
9. D. Morris, D. G. Evans, P. N. Green and C. J. Theaker, Object Oriented Computer System Engineering, Springer Verlag, 1996