An Object Oriented Design Method for Reconfigurable Computing Systems

Martyn Edwards and Peter Green
Department of Computation, UMIST, Manchester, United Kingdom
{M.Edwards, P.Green@co.umist.ac.uk}

Abstract

We present a novel method for developing reconfigurable systems targeted at embedded system applications. We show how an existing object oriented design method (MOOSE) has been adapted and enhanced to include reconfigurable hardware (FPGAs). Our work represents a significant advance over current embedded system design methods in that it integrates the use of reconfigurable hardware components with a systematic design method for complete systems. The objective is to produce an object oriented design methodology where system objects can be seamlessly implemented in either software or reconfigurable hardware.

1. Introduction

Embedded systems are proliferating in areas as diverse as high performance military applications and high volume consumer products [1]. The problem of managing design constraints is common to the development of almost all embedded systems, with factors such as cost, performance, size, power consumption etc. having a profound effect on the success of a product. Design constraints force developers into a detailed consideration of implementation options, since it is possible to realise different parts of a system in a variety of ways, each with different costs and benefits. For example, software may be written for a microcontroller or DSP, or custom hardware may be developed. In the hardware domain designers have the choice of an ASIC solution, or one based on FPGAs. Field programmable hardware solutions are becoming increasingly attractive [2] due to recent increases in logic capacities, and the ability of some devices to be partly or wholly reconfigured at run-time [3].

Design constraints typically relate to the system as a whole, and it is necessary to evaluate implementation options in the context of the complete system. There is, therefore, a need to represent the system design in an implementation-independent form to provide a context in which to evaluate the implementation choices against the design constraints. It is also crucial that a well-defined route exists from the implementation-independent model to the selected implementation technologies.

This paper is concerned with a development method for embedded systems that provides an implementation-independent modeling capability, and which facilitates implementation in any combination of software and application-specific hardware. The method, MOOSE (Model-based Object Oriented Systems Engineering) was initially developed to target software and/or ASIC implementations, and the primary focus of this paper is to examine the extensions that are necessary in order to derive implementations in programmable hardware.

2. Reconfigurable Embedded Systems

The use of FPGAs for the implementation of reconfigurable computing systems has been the subject of ongoing research since the early 1990s and a range of FPGA-based platforms is described in [4]. We are not concerned here with any particular hardware platform, but with the provision of methods and tools which allow the use of reconfigurable platforms to be integrated into a manageable embedded system development process.

At present, there is little, if any, previous work on design methods that are specifically targeted at supporting the development of systems containing reconfigurable hardware. Current reported methods tend to advocate the separate design and implementation of software and reconfigurable hardware, together with some form of run-time support [5, 6]. A key problem is that whilst software and hardware development are reasonably well understood, there is little work that applies to systems *as a whole*, especially when considering the integration of software and hardware, reconfigurable or otherwise.

Nevertheless, 'whole system' approaches have been proposed, including those that seek to integrate existing analysis and design methods to provide a seamless approach to the development of complete systems [7, 8]. Alternative paradigms devise homogenous methods and tools that provide an integrated route to system development [9, 10].

The MOOSE method adopts a homogenous approach based on object oriented (OO) principles. The arguments

supporting the use of OO techniques for software are well known [11] and MOOSE extends classical OO principles to apply to hardware as well as software. Previous work [9], and that of others [12, 13] has shown that OO principles may be applied with advantage to both *complete systems*, and to fixed function hardware alone. Hence it is a natural step to extend MOOSE to model system elements containing reconfigurable hardware.

3. The MOOSE Method

The approach starts with an *Informal Product Specification*, which is formalised into a complete specification through the development of the *Behavioural Model* (BM). This model is a graphical, hierarchical representation of the system objects, showing how they collaborate to satisfy the functional requirements. Complex objects are decomposed into sets of simpler objects until its 'primitive' objects are identified. At this stage objects are *uncommitted* to either hardware or software implementations.

Primitive object behaviour is specified in the next phase, both graphically and by adding C++ code. This renders the model executable so that its behaviour can be validated against the functional requirements. This *Executable Model* (EM) is tested against a number of scenarios [9], and when its behaviour is satisfactory, it represents a validated architecture that satisfies functional constraints.

The implementation technology for each object is identified, via a 'Transformational Codesign' process, using 'time-aware' model execution that supports limited timing/performance analysis. Additionally, the external interface of a system is completed, the numbers and types of processors are determined, and concurrent threads are analysed. The resulting *Committed Model* (CM) satisfies the system functional requirements and the design constraints. Additional hardware and software objects, for example interrupt controllers and operating system components, are normally required to support the operation of a system. A *Platform Model* (PM) is then developed which incorporates these additional objects.

In the final stage, source code is synthesised from the *Committed* and *Platform Models*. C++ code is produced for each software object, and VHDL component specifications are developed [13] for hardware objects

4. Hardware Objects and Run-Time Support

The concept of an object as an entity encapsulating behaviour and state [11] is as applicable to reconfigurable hardware as it is to software. This is also true of the notion of class as an abstraction defining the behaviour and state shared by a set of objects. In terms of reconfigurable hardware, behaviour is provided by the configuration, and state is held either within the configuration (for example in registers), or in external memory. A class definition is simply a description of the configuration, along with the definition of a scheme for storing object state. Basic inheritance may, therefore, be implemented in a straightforward fashion. Dynamic object creation and destruction can, in principle, be catered for by dynamic reconfiguration of a programmable array. The implementation of polymorphism is rather more difficult and requires significant software support. However, whilst the realisation of advanced OO concepts in reconfigurable hardware continues to be of research interest, a relatively straightforward view of objects exhibiting behaviour and containing state is initially sufficient in MOOSE.

The significant differences between software and hardware objects are most obvious in connection with communications mechanisms, since the usual OO software message passing communication does not normally provide a satisfactory abstraction of communication between hardware elements [9]. More general communications mechanisms must be introduced that seek to model hardware as well as software. This is recognised in MOOSE where a number of mechanisms are already provided for modeling hardware communications [9].

If reconfigurable hardware is to be exploited in embedded systems, run-time software support must be provided. We envisage an FPGA run-time support system (RTSS), which runs on a conventional processor, and is responsible for configuring the array, supporting advanced OO facilities, and scheduling object execution.

5. Extending MOOSE for Reconfigurable Systems

The BM and EM are independent of technology, and so no changes are needed at the early stages of the MOOSE method. In the development of the CM, a notation must be provided for specifying which objects are to be implemented in reconfigurable hardware, the identity of the FPGA on which they will run, and the processor to which the FPGA is interfaced. The structure of the PM must be modified to include facilities for the scheduling of hardware objects on to the FPGAs. It must also provide for communications between objects running on the FPGAs and the rest of the system.

The extensions to the MOOSE method are illustrated by an example. The application is a Video Surveillance System (VSS) [15], which is designed for use in buildings and consists of a set of Embedded Video Controllers (EVCs) and a single Supervisor Control Centre (SCS). The EVCs are distributed in a building for capturing, compressing, buffering, and transmitting images over a network to the SCS for decompression and display.

5.1. MOOSE Committed Model

The complete CM is too complex to be described here in detail, hence only a small relevant portion is presented which concentrates on the EVC alone. Figure 1 shows a simplified refinement of a higher level **EVC** object, with three software objects (**Camera Manager**, **Frame Manager**, and **Compressed Images**) sharing an ARM processor, and one hardware object (**Compressor**) running on FPGA1 interfaced to the ARM processor. The EVC operates in a number of different modes under the control of the **Frame Manager**, which receives commands from the SCS. Full details of the MOOSE notation are given in [9], but see also Figure 2.

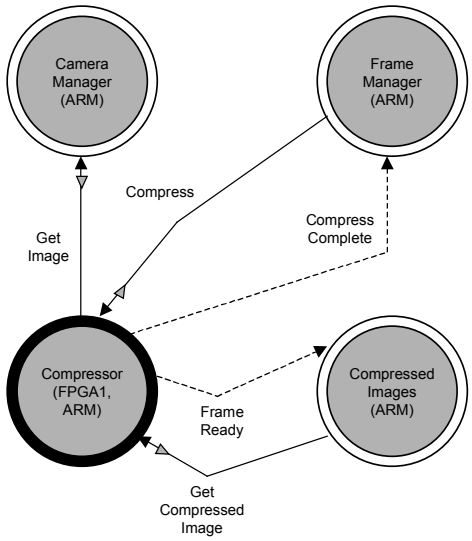

Figure 1. EVC: committed model

Figure 2 shows the refinement of the **Compressor** object as a set of hardware objects, the design being adapted from the "Versatility" benchmark in [15].

5.2. The MOOSE Platform Model

A MOOSE PM represents a system's execution environment, including the system software and hardware. At the highest level, the platform is represented as a single object that interfaces with external objects, which represent hardware devices that are outside the system. There is also an external object for each processor in the CM which represents the applications software associated with that processor. Communications between these objects and the platform represent the services that the platform performs on behalf of the application software.

A PM is itself composed of three objects: communications view (CV); software interface view (SIV); and hardware view (HV). The CV provides a high-level communications interface between objects running on different processors. The SIV provides a software scheduling mechanism together with details of the hardware/software interface, whereas the HV describes the platform hardware.

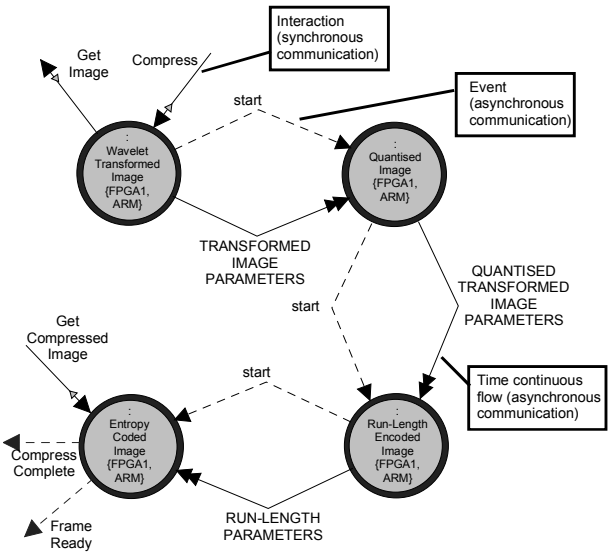

Figure 2. Compressor: committed model

In extending MOOSE to handle reconfigurable hardware, we distinguish between an object (hardware configuration) and the device on which it runs - analogous to the relationship between software and the processor that executes it. Hence, an external object is inserted into the PM for each FPGA in the system, representing the hardware objects that are configured to run on the FPGA.

5.2.1. The Communications View

The CV supports communication between objects running on different processors. The CM shows communications between such objects as logical connections (Figure 1). The purpose of the CV object is to map logical communications from the CM to the communications mechanisms provided by the platform. The 'internals' of the object then make use of the low-level platform software, in the SIV, to perform the actual communication.

The CV typically contains a number of simpler objects, which act as *proxies* [16] for the remote objects. To facilitate communication between FPGA objects and software, and to retain the approach to platform modelling a communication object is added to the CV to service FPGA objects. This will execute on the FPGA, and will receive internal "calls" from its resident configurations and map them to the hardware communications mechanisms provided by the platform. Note that it is

possible to synthesise this object's interface automatically.

Figure 1 also shows logical communications between FPGA objects and software (the *Frame Ready* and *Compress Complete* events). These are intended to trigger activity in the related software objects, for example, *Frame Ready*, causes the **Compressed Images** object to load a compressed image into an internal queue. This event is raised by a child of Compressor (**Entropy Coded Image**), and is sent to the FPGA communications object in the CV. This maps the event to an interrupt for the ARM processor via the FPGA output interface. The associated interrupt handling software invokes an operation within the **Compressed Images** object that results in the call *Get Compressed Image* being issued.

Software-to-FPGA object communication must also be handled. For example, consider the *Compress* interaction issued by **Frame Manager** in Figure 1. This call causes **Wavelet Transformed Image** to read and compress a new image. Depending on the status of the actual implementation, this object may not be resident on the FPGA at that instant, and if this is the case, the FPGA must be reconfigured to contain this object. Software-to-FPGA object communications of this kind fit neatly into the MOOSE platform modelling framework, whereby calls from software objects to FPGA objects are directed to the CV. The appropriate part of the CV then maps the application-specific call to low level communications encapsulated within the **Software Interface View** object. Figure 3 shows a simplified version of the of the ARM platform, where this mapping is indicted by the bundle (a bundle aggregates a number of related communications) *RTSS COMMUNICATIONS SERVICES*. The FPGA RTSS would be invoked to reconfigure the FPGA to contain the new object. The other bundle, *RTOS COMMUNICATIONS SERVICES*, allows communication between software objects on different processors.

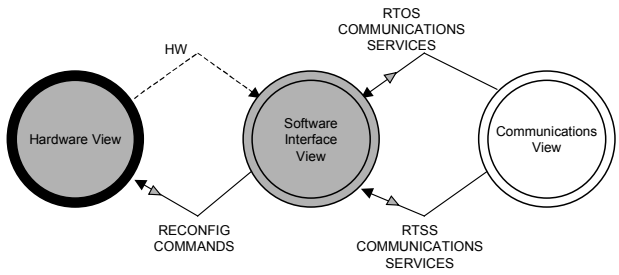

Figure 3. ARM platform: high-level view

5.2.2 The Software Interface View (SIV)

The SIV (Figure 4) contains software objects that control hardware, including a scheduler for concurrent software and hardware device driver objects - a real time operating system (RTOS) may be used here. For systems with reconfigurable components this software is supplemented with a RTSS that manages the FPGAs.

The **Virtual RTOS** object provides an abstraction layer between the application software and the RTOS itself. The **RTOS** and **RTSS** are shown as non-primitive objects that contain scheduling mechanisms and interface objects. The interfacing of software to fixed-function hardware is performed by interface objects [9], which appear in the refinement of **RTOS**. The **RTSS** contains analogous configuration drivers to facilitate the communication between software and the FPGA.

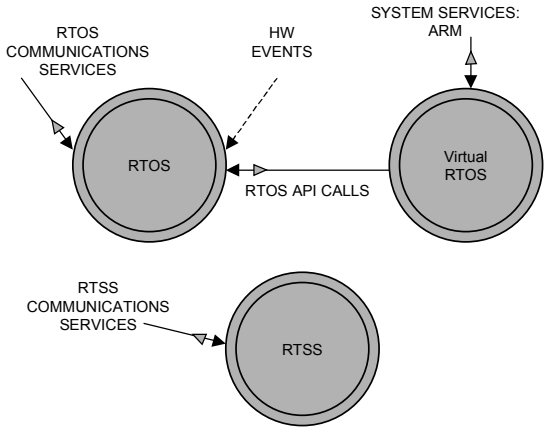

Figure 4. Software interface view

Requests for communication between software and FPGA objects are routed through the appropriate communications object, which interacts with the **RTSS** to check whether communication is possible (for example, to confirm that **Wavelet Transformed Image** is currently active). If so, the **RTSS** allows the call, via a configuration driver, and interacts with the FPGA across the bus. If the target FPGA object is not resident, the call from software is normally suspended by the RTOS and resumed when the **RTSS** object has reconfigured the array to allow the target object to be executed.

FPGA objects will typically communicate with software objects via interrupts. For example, the *Frame Ready* event generated by the **Compressor** (Figure 1) would typically be implemented as an interrupt (mapped to part of *HW EVENTS*) handled by a service routine in the RTOS. The routine would invoke the *Frame Ready* operation within **Compressed Images**. An implementation would utilise a thread within this object which would be suspended during compression, but resumed by the *Frame Ready* interrupt handler. The thread reads the image, via the *Get Compressed Image* interaction, utilising the above platform mechanisms.

5.2.3. The Hardware View (HV)

The HV defines the hardware topology of the system.

The initial form of the HV can be synthesised from the CM [13] and consists of the processor and any fixed-function hardware objects that are interfaced to it, plus any FPGAs. This initial model is developed further, via the addition of physical interconnections between devices (typically one or more buses), and supporting hardware objects such as an interrupt controller and bus arbiter. A full system implementation may be developed through the addition of further detail [13] using standard hardware design techniques, including synthesis from VHDL descriptions of hardware objects. One version of the platform model for the VSS is shown in Figure 5 and includes the **FPGA1** and its **Configuration Store**. Device reconfiguration is controlled by the RTSS software via the bundle *RECONFIG COMMANDS*.

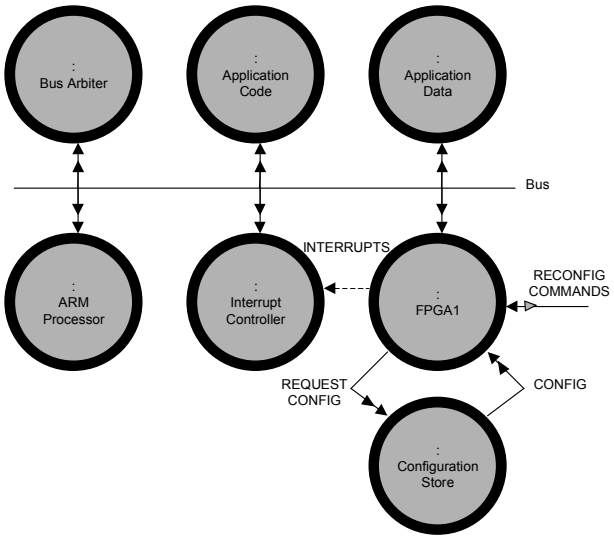

Figure 7. Hardware view

6. Conclusions

We have presented an approach (MOOSE) to the development of *complete* embedded systems containing software, fixed-function hardware, and dynamically reconfigurable FPGA components. The uniform framework provides a context in which to evaluate different implementation options on the basis of design constraints imposed on a project. Clear and well-defined routes into implementation are provided, which are supported by synthesis and the potential to reuse existing objects. The method also provides a way of reasoning about the behaviour of the execution environment and the interaction between the application-specific part of the system and the platform.

Further enhancements to MOOSE will include consideration of reactive systems where reconfigurable hardware-based objects can be scheduled in response to events from a system's external environment. In many respects this is analogous to a RTOS that manages prioritised tasks or threads and includes the idea of 'object pre-emption' which involves saving/restoring the state of pre-empted hardware-based objects.

Finally, we believe that our unified approach to system development will permit the further use of OO methods for embedded systems. Designers will be able to use such techniques without being specifically concerned with whether an object is implemented in hardware or software – a situation that is not true with many currently available methods and tools.

7. References

[1] Osterberg, L, "Standards in real-time systems: enabling market growth", in *Business and work in the information society: new technologies and applications*, IOS Press, 1999.

[2] M.D. Edwards and J. Forrest, "Software acceleration using programmable hardware devices", *IEE Proceedings - Computers and Digital Techniques*, 1996, **143** (1) pp.55–63.

[3] "Virtex 2.5V FPGA series (XCV00)", Xilinx Inc, 1998.

[4] M.D. Edwards, D.G. Evans, and P.N. Green, "Platforms for reconfigurable computing", Workshop on Reconfigurable Computing, PACT '98, October 1998, Paris, France, pp. 17-21.

[5] J. Fleischmann, K. Buchenrieder, and R. Kress, "A hardware/software prototyping environment for dynamically reconfigurable embedded systems", 6th International Workshop on Hardware/Software Codesign, March 1998, Seattle, USA, pp. 105-109.

[6] P.I. Mackinlay, P.Y. Cheung, W. Luk, and R. Sandiford, "Riley-2: a flexible platform for codesign and dynamic reconfigurable computing research", 7th International Workshop on Field-Programmable Logic and Applications, September 1997, London, England, pp. 91-100.

[7] Kronloff, K., *Method integration: concepts and case studies*, John Wiley & Sons, Chichester, 1993.

[8] Thome, B., *Systems engineering: principles and practice of computer-based systems engineering*, John Wiley & Sons, Chichester, 1993.

[9] Morris D., D.G. Evans, P.N. Green, and C.J. Theaker, *Object oriented computer system engineering*, Springer Verlag, Berlin,1996.

[10] N.S. Woo, A.E. Dunlop, and W. Wolf, "Codesign from cospecification", *IEEE Computer*, 1994, **27** (1) pp 42-47.

[11] Booch, G., *Object oriented design and analysis*, Benjamin/Cummings, 1994.

[12 W. Wolf, "Object oriented co-specification for embedded systems", *Microprocessors and Microsystems*, 1996, **20** (3) pp. 141-147.

[13] D.G. Evans, P.N. Green, D. Morris, and P.B. James-Roxby, "A systems approach to embedded system development", in *Embedded microprocessor systems*, IOS Press, 1996.

[14] P.N. Green, "MOOSE models of the video surveillance system", ESPRIT Project 20.592 (OMI/MODES), Deliverable TR6.2.5, October 1998.

[15] "Benchmarking tools and assessment environment for configurable computing: benchmark specification document – versatility stressmark", Honeywell Technology Center, 1997.

[16] Douglass, B.P, *Real time UML*, Addison-Wesley, 1998.