The Modelling of Embedded Systems Using HASoC

P. N. Green and M. D. Edwards
Department of Computation, UMIST, Manchester, United Kingdom
{peter.n.green, m.edwards@co.umist.ac.uk}

Abstract

We present a design method (HASoC) for the lifecycle modelling of embedded systems that are targeted primarily, but not necessarily, at SoC implementations. The object-oriented development technique is based on our experiences of using an existing modelling technique (MOOSE) and supports a lifecycle that explicitly separates the behaviour of a system from its hardware and software implementation technologies. The design process, which uses a UML-RT-based notation, begins with the incremental development and validation of an executable model of a system. This model is then partitioned into hardware and software to create a committed model, which is mapped onto a system platform. The methodology emphasises the reuse of pre-existing hardware and software platforms to ease the development process. An example application is presented in order to illustrate the main concepts in HASoC.

1: Introduction

Today's SoC devices contain tens of millions of transistors and in a few years time are predicted to contain well over 1 billion transistors [1]. These rapid advances in process technology have led to an increasing productivity gap between the number of transistors available and the number of transistors that can be integrated in a design – annual growth rates of 58% and 21%, respectively, have been widely reported [1]. This gap is, in part, due to the lack of power in current methods and tools to permit the design, verification, and implementation of increasingly complex systems within reasonable timescales. A major challenge is to produce innovative design methodologies that go some way to reducing this productivity gap. We address this challenge in this paper.

In order to meet shortening time-to-market constraints and reduce the astronomical development costs for new complex SoCs, it is recognised that the design process must make increasing use of reusable hardware and software components (IP). It is hard to imagine a completely new design that does not use existing IP to some extent. Incremental design procedures, where ~ 25% of the final system contains newly developed IP, will become commonplace [1]. This will lead to the development of pre-characterised *platform* architectures [2] that are essentially the same for a specific class of applications, for example, wireless communications [3]. It is likely, however, that a platform will need to be customised in order to meet the requirements of a particular instance of the application space by, for example, adding, removing, replacing, or tweaking the parameters for IP components [4]. The implication is that an SoC development methodology will need to support an incremental design strategy using existing platforms.

The evolution of a system specification, capturing requirements and constraints, hardware/software co-design, design space exploration and partitioning, development of hardware and software architectures, and integration of IP are major design challenges that could be incorporated into a single, consistent design methodology. In our research, we have begun the development of a such a methodology, HASoC (Hardware and Software Objects on Chip) that employs an object-oriented approach to the development of complete embedded systems (including hardware, software, and platform architectures). HASoC is a fusion of our existing MOOSE approach [5] with the UML notation [6], coupled with an iterative, incremental approach to system development.

The next section identifies our experiences with the MOOSE method, indicates the requirements for our new methodology. It also identifies the required extensions to UML for use within the context of our work. Section 3 introduces a "typical" embedded system that could be implemented as an SoC and which is used as a working example in the remaining sections of the paper. An overview of the HASoC modelling flow is discussed in Section 4. The final section summarises our work to date and presents our plans for future research.

2: Embedded System Development Methods

In recent years, a number of noteworthy design methodologies have emerged for the development of real-time systems that include facilities for requirements capture and analysis, design, implementation, and testing [7, 8, 9]. Whilst most of this work has been targeted at the development of real-time software, some researchers have proposed similar frameworks for the co-development of hardware as well as software [10].

Our approach to the development of SoCs is founded on past research and experiences within our MOOSE object oriented hardware/software design environment [5]. The MOOSE method starts with an abstract behavioural model of a system whose functionality is validated through execution of this model. Subsequently, the model is partitioned into appropriate hardware and software technologies (committed model) so that the product's design constraints and satisfied and its design optimised. The MOOSE method has been successfully applied to the implementation of a number of embedded systems [11].

Executable modeling has proved to be a very valuable method for understanding a system’s behaviour, and enabling system functionality to be verified early in the development cycle. It has been demonstrated that this capability significantly reduces the detailed implementation work required in the later stages of the development lifecycle. However, experience with the MOOSE approach has highlighted a number of problems that restrict its usefulness in a wide variety of applications.

The strictly sequential nature of the lifecycle causes difficulties, where the development of the complete executable model must be finished before the committed model can be generated. MOOSE has proven to be unresponsive to changing product requirements during the development process. A rigid top-down approach to model development was adopted, which has been shown to be not only difficult to apply [12] but also tends not to support the reuse of hardware and software components. The development of the system platform in MOOSE is delayed until after the system objects have been committed to hardware or software implementations. Whilst this has the praiseworthy objective of allowing a system platform to be tailored specifically for an application, it also runs counter to the incremental/reuse design procedures that have been identified as being necessary in the future. MOOSE employs a proprietary graphical notation to express object-oriented designs, which restricted its widespread acceptance in the design community. Lately considerable attention has been focused on the Unified Modeling Language (UML) [6], which is rapidly becoming the de facto object-oriented modeling language [13], and for this reason UML has been chosen as the notation for HASoC.

2.1: Required UML Extensions

UML is a notation that has achieved the status of a standard modeling language for object-oriented software development. In its basic form it is applicable to a wide variety of systems, but lacks specialist concepts for particular application domains. Hence a number of specialist extensions have been proposed, including one for real-time systems, UML-RT [14], which we consider to be applicable to hardware as well as software. On the basis of our work and experiences with MOOSE, we believe that it is necessary to put forward further extensions to the UML-RT proposal to make the notation more suitable for mixed technology embedded systems. Other researchers have reached a similar conclusion [15].

The UML-RT notation emphasises a hierarchical system organisation via *capsules*, which have a specific execution and communication semantics. From a behavioural perspective the internal functions of a capsule are represented by a single statechart. From a structural point of view, capsules may contain sub-capsules as well as, or instead of, the statechart. Capsules communicate via owned objects called *ports* that implement specific *protocols*. Protocols are named sets of signals communicated in a point-to-point fashion between capsule instances. Finally, *connectors* between ports in different capsules indicate message communications paths.

The execution model is based on the transmission and reception of signals through ports that cause state transitions within a capsule’s statechart. Communication is typically asynchronous, and ports contain message queues. A capsule’s thread of control chooses a message from one its ports' queues and dispatches it to the state machine, which takes the appropriate actions and transitions. Once a capsule has started to process a message, it will run-to-completion before responding to another message. However, the execution of a capsule may be pre-empted by another capsule.

Whilst UML-RT is clearly applicable to a subset of real-time software systems, we believe it is equally clear that extensions are necessary if it is to be used successfully in the development of complete embedded systems, including software, hardware, application, and platform. The need for generalisation arises from the fact that not all aspects of embedded systems are well represented by an event-driven model – a significant number of applications are, for example, essentially data-driven rather than event-driven [16, 17].

We have already extended the basic UML-RT model to obtain a data-driven capsule whose ports can receive streams of data and process them in the capsule’s thread of execution [18]. This mechanism is based on the "time-continuous flow" communication model in MOOSE, and has parallels with one of the abstract mechanisms proposed for Layer 1.0 of the VSIA Virtual Component Interface Model [19].

3: Example System: Biometric Access Control

A door access control system, based on biometrics, will be used as an example throughout the rest of this paper to illustrate the design approach adopted in HASoC, and the application of UML to the development of complete embedded systems. The chosen example is a distributed system for use in secure environments and consists of a set of door units and a central administrative server. The door units provide secure access, based on iris recognition [20]. Individuals who are to be granted access to secure areas are registered with the system. This requires the capture of iris images, the extraction of iris codes, and the storage of codes in a secure, remote database located on an administrative server. In this paper, we will concentrate on the development of a single door access system, and only briefly touch on the operation of the server system – full details of this work can be found in [21].

When an individual wishes to enter a secure area he/she looks into a wall mounted camera unit. The user indicates that they are ready; the system captures an image, and informs the user of the progress of the access request and the door status. The iris code is extracted from the image, encrypted, and sent to the server, where it is decrypted and compared with iris codes stored in the secure database. If a match is found a message is sent to the door unit instructing it to open the doors, otherwise a message is sent back indicating that the door should remain locked.

4: The HASoC Design Lifecycle

The review of our current system development method has led us to propose a new design lifecycle for embedded SoC devices. In doing so, we have attempted to retain the benefits of our MOOSE method, whilst eliminating those features we know to be unsuitable and problematical. The method is known as HASoC, and it introduces an iterative, incremental lifecycle, which enhances the concepts expressed in the "Rational Unified Process" [6, 22]. Figure 1 illustrates the proposed HASoC lifecycle.

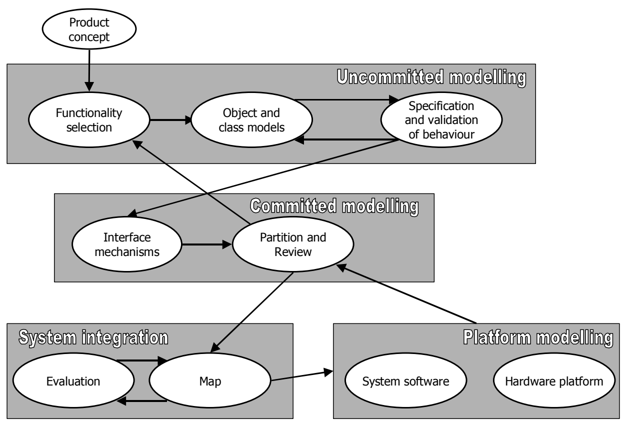

Figure 1: HASoC Lifecycle

4.1: Product Concept

This initial stage informally defines the scope of the product as an *ad hoc* set of statements, in a natural language, which are not necessarily complete. The aim is to document what is known about the product, particularly in terms of its functional requirements, design constraints, and system platform.

4.2: Uncommitted Modelling

In the *functionality selection* phase the functional requirements are analysed in a number of stages. The first stage develops the core functionality of the system, with the later stages adding additional functionality as and when necessary [22]. Decisions about what constitutes the core functionality will be application specific, and may be taken on the basis of a variety of criteria. For example, some aspects of functionality are intrinsic to the purpose of the product and may influence when they are chosen for development – capturing an iris image in the biometric access system would be a core activity. Other factors could involve perceived risk, for example, an element with a high complexity and for which no reusable Intellectual Property (IP) is available should be included at an early stage – extracting the iris code, for example. The expertise of the design team would strongly influence how and when the many activities in a system were considered.

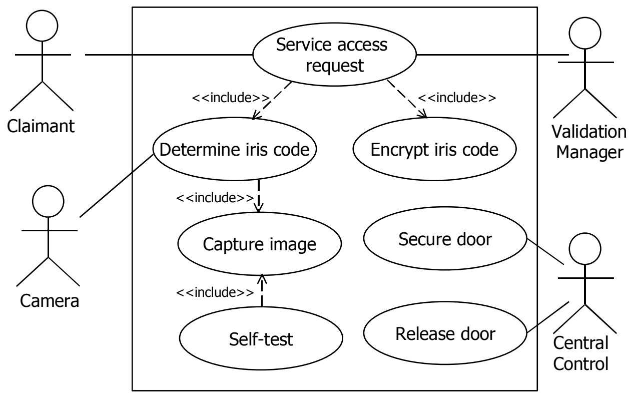

Figure 2: Use Case Model for a Door Unit

A use-case-driven approach is a convenient method for not only exposing the required hardware and software functionality of a system [23], but also as a basis for incremental development. Since there is a well-defined route from use cases to object models, they are an extremely attractive option at this stage. Subsequent iterations of this design phase enable further emerging requirements to be scheduled in the development cycle.

Figure 2 gives a UML use case model for a door unit of the biometric access system. The model consists of seven use cases, which capture the high level user-functional requirements.

In the *objects and class models* phase, these models are developed for each selected use case(s). A typical approach would be to develop sequence diagrams for key use case scenarios, and then to synthesise the UML class and capsule collaboration diagrams from the sequence diagram.

A sequence diagram for a successful *Service access request* scenario is shown in Figure 3. The diagram indicates that the door controller, vision system, image, matcher, and door are all active objects (capsules). The camera and lock are passive objects that act as high-level interfaces to the camera and lock drivers, which are part of the platform model. Once an image has been acquired, the vision system processes it by creating an active image object. An iris object is subsequently created which performs the detailed computation of the iris code. This code is subsequently returned to the vision system and then sent as a synchronous message to the door controller.

A possible capsule collaboration diagram for the door controller is given in Figure 4. The diagram indicates that the vision system and door capsules are realised as sub-capsules within the door controller itself, thus allowing them to be active concurrently. In fact, the identification of concurrency is an important consideration when generating a capsule collaboration diagram from a sequence diagram. Note also that ports are included in the door controller capsule to allow communication with the user and the remote iris code matcher.

A UML class model depicts the static structure of the application-specific part of the system, and the corresponding UML object model shows the run-time behaviour of instances of the classes, as they collaborate to deliver the functionality under development in the current iteration cycle – neither diagram is shown here. The construction of the object and class models normally uses standard development techniques, for example, the derivation of sequence diagrams and capsule collaboration diagrams from use case scenarios [24]. Both the class and capsule models provide important information for the synthesis of the executable model: the class model supports the synthesis of class interface code; and the sequence and collaboration diagrams provide information about run-time inter-object communications. From these models a skeleton executable model can be synthesized that supports the system-wide validation of object-level communications. In order to provide a finer degree of behavioural validation, detailed code (for example, in C++) would be added to the skeleton executable model in the *specification and validation of behaviour* phase of the lifecycle.

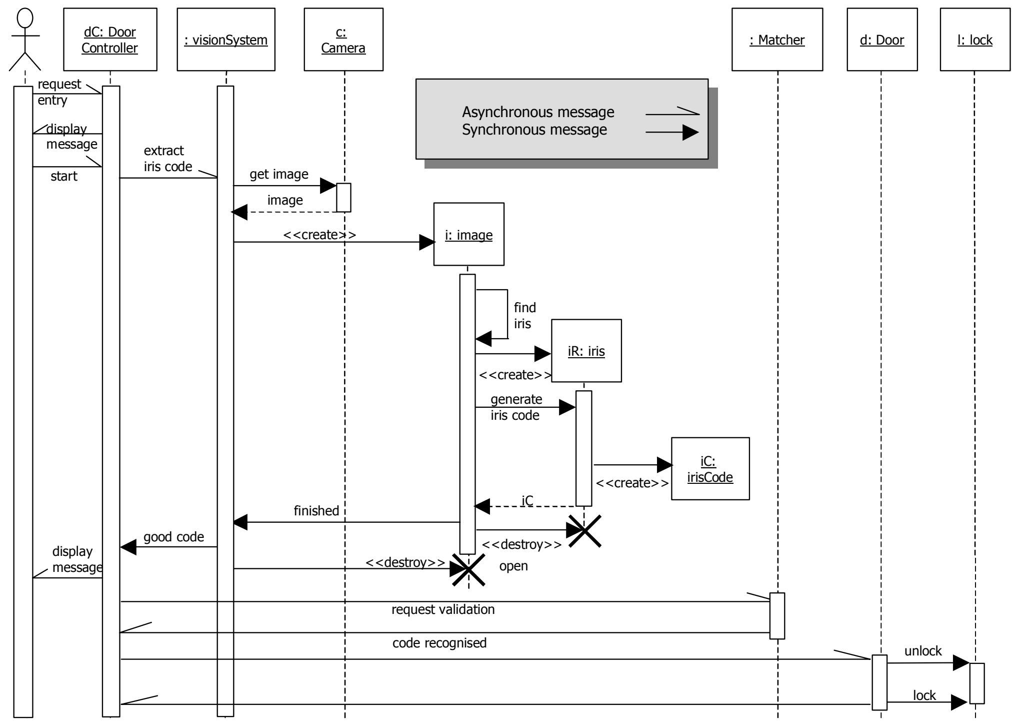

Figure 3: Sequence Diagram for the Successful *Service access request* Scenario

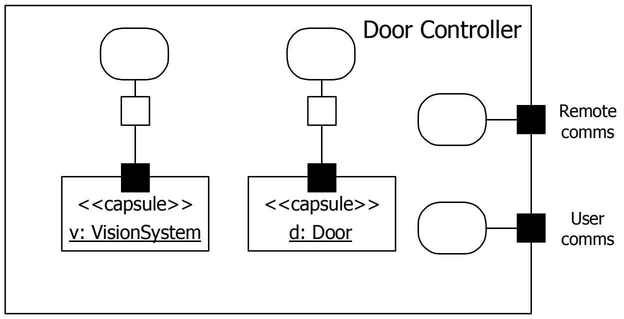

Figure 4: Capsule Collaboration Diagram for the Door Controller

The use cases and their scenarios can be used to build up a test programme for the executable model. At some point during this phase the executable model from the current iteration is integrated with the one produced in the previous iterations. This will typically occur during the detailed development of the new executable model, since at least some of the classes/objects will be common to different iterations.

Further information concerning the detailed development and use of executable models can be found in [5].

4.3: Committed Modelling

In generating a committed model, we are principally concerned with partitioning the abstract executable model into hardware and software implementations, and then allocating the resulting 'committed capsules' to the processing elements in the platform. The first sub-phase considers the external interface of the system. In some projects, interaction with the environment will be predefined at the start of the project, whereas in others it must be determined as part of the design process. In the biometric access system, design decisions must be taken about how to support human interaction, for example, whether commands and responses should utilise either voice input and speech output, or a keypad and LCD display. Under such circumstances, additional objects must be added to the model to support such design decisions.

The *partition and review* stage includes a number of activities and is dependent on whether or not a platform description exists. Partitioning is performed at the object level, with some objects being identified for software implementation, with the others implemented in hardware. Our method is not prescriptive about how partitioning is to be achieved: manually and/or automatically. Since an executable model is available, run-time performance measurements, analysis of the code, or examination of the communication patterns observed during model execution can be employed to guide the partitioning process. Once the partitioning decisions have been taken, the model must be reviewed, to ensure that the inter-object communications mechanisms are consistent with the chosen object commitment and the identified concurrent activities have been correctly implemented.

The first sub-phase of the commitment of the door controller is illustrated in Figure 5. Our overall approach, in this example, will be that reactive elements are committed to software and compute-intensive capsules are committed to reconfigurable hardware, in particular those concerned with processing an iris image and extracting the iris code. (This decision was made in an *ad hoc* manner in order to illustrate the commitment process and notation used). Note that we have already developed a prototype hardware implementation of these components, which are specified in Handel-C and implemented on a Xilinx Virtex FPGA [21].

Figure 5 indicates that the commitment process has introduced a new capsule *u:UserPanel*, which supports user interaction with the system, and will contain hardware and software objects that service a keypad and an LCD display. We assume that the other capsules *v:VisionSystem* and *d:Door* are also composites, whose decompositions contain hardware and software components, and so no commitment is indicated at this stage. However, we have already decided that a capsule's state machine will be committed to software. This is indicated in the diagram by extending the UML-RT notation to identify such software assignments: *<<sw P1>* that states that the state machine is executing in software on processor *P1*.

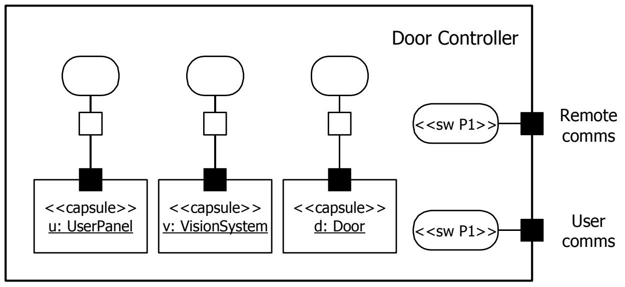

Figure 5: Commitment of the Door Controller – First Phase

Figure 6 indicates a possible decomposition of the *v:VisionSystem* capsule. Its state machine is committed to software on processor *P1*, whilst the child capsule *I: image* is committed to hardware. Again, we have extended the UML-RT notation to identify such hardware assignments: the capsule is implemented on FPGA *F1*, which is closely coupled to processor *P1*. The commitment of the *u:UserPanel* and *d:Door* capsules would be undertaken in a similar manner.

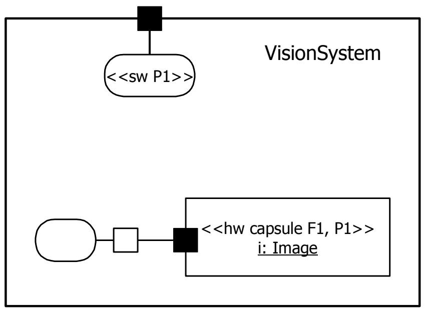

Figure 6: Commitment of the *Vision System* Capsule

4.4: Platform Modelling

All elements of the abstract model are committed to hardware and software on the basis of design constraints relating to, for example, performance and availability of IP elements. In order to construct a working system, however, consideration must be given to those additional components that are required to provide the execution environment, for example, processors, buses, memories, the RTOS, and device drivers. The system requires a *hardware platform* to execute the committed model, together with a significant level of *system software* support. Both these elements make up the *platform model*.

The hardware platform is an architecture that satisfies the hardware and software constraints of the application. At one end of the spectrum, a pre-existing hardware platform may be used without change. It is, however, more likely that an existing platform would be customised for a particular application through the addition, removal, replacement, or "tweaking" of hardware and software objects. In each design iteration, the capabilities of the platform may be updated to reflect the additional functionality introduced into the system.

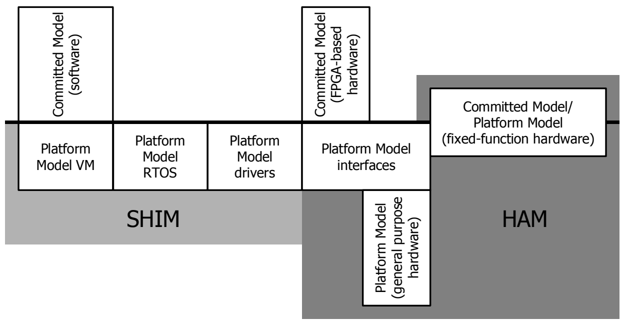
Committed Model (software)
Committed Model (FPGA-based hardware)
Committed Model/ Platform Model (fixed-function hardware)
Platform Model VM
Platform Model RTOS
Platform Model drivers
Platform Model interfaces
Platform Model (general purpose hardware)
SHIM
HAM

Figure 7: Relationships between SHIM and HAM Models

The platform model is concerned with describing the overall execution environment in sufficient detail to facilitate the implementation of the complete system. In our case the platform model has two major elements: the Software-Hardware Interface Model (SHIM), and the Hardware Architecture Model (HAM). The relationships between these two models and their various internal components are shown in Figure 7.

The SHIM consists of a virtual machine (VM), the RTOS, and device drivers that facilitate communication between hardware and software capsules. The VM provides the interface between the software part of the committed model and the system platform, in particular the RTOS. The VM interface can be inferred from the committed model. For example, consider the direct communication from a software capsule to a hardware capsule as shown in Figure 8(a). In this case, the direct connection of the committed model is implemented by the platform. At the top level, a software proxy exists for the hardware object; see Figure 8(b). This is a conventional passive object that essentially implements an interface that is the conjugate of the software capsule. The proxy has the responsibility for receiving a call from the software capsule's port. The proxy forwards this call, through the RTOS API to the appropriate device driver, which communicates with the physical hardware. Both synchronous and asynchronous communication can be achieved in this fashion. If a driver is not available for the hardware capsule then one must be developed based on the conventions employed by the RTOS and the interface of the hardware object. The driver must subsequently be linked to the RTOS. The SHIM, therefore, typically consists of the UML component models for both the RTOS and the drivers.

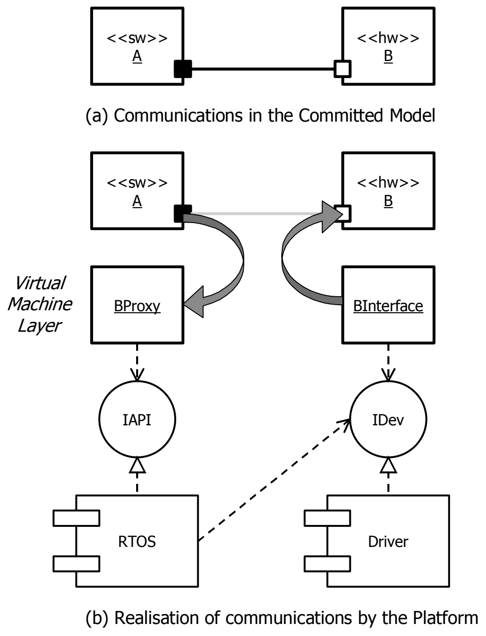
<<sw>>
A
<<hw>>
B
(a) Communications in the Committed Model
<<sw>>
A
<<hw>>
B
Virtual Machine Layer
BProxy
BInterface
IAPI
IDev
RTOS
Driver
(b) Realisation of communications by the Platform

Figure 8: Communications between Committed Hardware and Software Objects

The HAM is initially based on a UML deployment diagram, which is used in software projects to represent the underlying hardware as a collection of computational resources. Within HASoC, deployment diagrams play a similar role and for pre-existing platforms they are trivial to construct. For the door access system, a partial deployment diagram is shown in Figure 9. The diagram is useful for indicating the basic structure of the platform and can show where particular software components are executed, as well as where and how IP components are connected to the system.

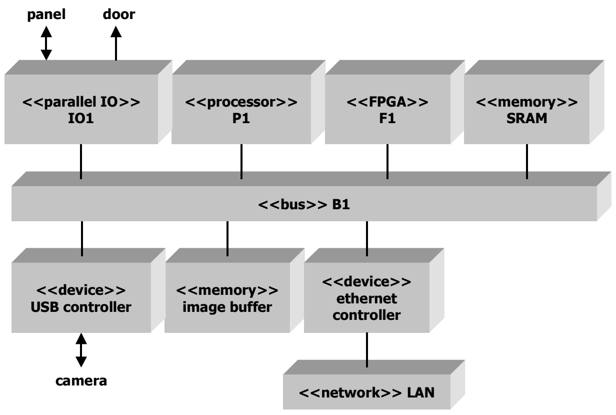

Figure 9: Partial Deployment Diagram for the Door Access System

The representation of the basic components and their topology forms a useful starting point in the development of the HAM. From the deployment diagram it is possible to derive a further object model, which not only represents the hardware of the system but also supports its synthesis and simulation. Such a model provides a convenient framework for organizing the platform definition, enabling elements to be added, removed, replaced or modified. The objects in the model are IP components that are assumed to have synthesisable functional descriptions but may not be able to interface to other blocks without, for example, bus wrappers.

Capsule notation is used to indicate that the objects are concurrently active – see Figure 10. The port-based communications are based on abstractions of hardware signals that employ a known set of protocols. In our case, the ports effectively implement the communications interfaces specified in the VSIA definition [19]. Such interfaces are usually defined at various levels of abstraction for simulation and synthesis purposes, for example, application level, system level, virtual component level, and physical level. Techniques for refining application level descriptions to physical level ones have been proposed in order to connect hardware and software IP components [26]. This model can also form the basis of a co-simulation environment if appropriate executable models of the IP components are available. The hierarchical interface structure will readily allow the inter-working of different executable models. We have not yet, however, explored this possibility within the HASoC framework.

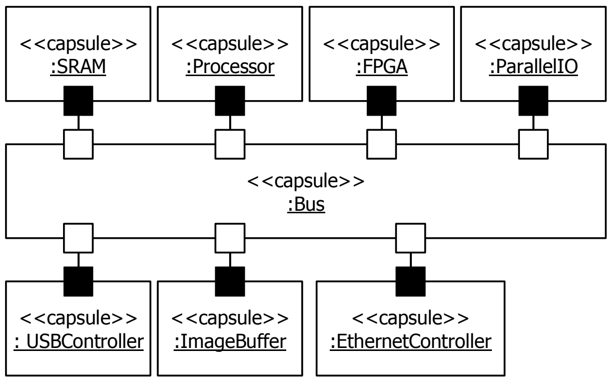

Figure 10: Hardware Architecture Model for the Door Access System

4.5: System Integration

System integration involves exploring the implementation space for a given application in order to find a suitable platform for a set of design constraints. The integration process *maps* a committed model onto the current incarnation of the platform model. The *evaluation* process is effectively a performance assessment of the system in order to determine whether or not the current platform can execute the current committed model in a functional sense and satisfy the design constraints. The results of the integration process can be used to reconfigure the platform and/or modify the committed model (in terms of allocation of objects to hardware and software) so that the constraints are satisfied. System integration can be performed at multiple levels of abstraction, from a purely functional level to a detailed clock cycle accurate timing level as defined by the platform model. This is supported in HASoC through iterative mapping, evaluation, and platform development. In all cases, the system integration process attempts to 'execute' the committed model on the hardware architecture model.

5: Conclusions

In proposing a new lifecycle for the development of embedded systems-on-chips, we have analysed our experiences with our existing MOOSE approach. Although MOOSE has significant virtues, a number of difficulties are apparent when applying it to real problems. In particular the need to complete each design stage in a sequential manner, with no iteration loops, makes the method unresponsive. Moreover, the generation of the system platform and the mapping of the application-specific committed model on to it take place comparatively late in the development cycle, thus, increasing risk. The use of a custom notation is also a considerable drawback to the acceptability of the methodology.

All these factors have been addressed in the HASoC lifecycle. By basing the notation on UML, we reap the benefits of using a widely understood language. By using ideas of iterative system construction, we have developed a lifecycle that will be more responsive to changing requirements, and where work on the underlying platform can commence much earlier and proceed concurrently with the iterative development of the application-oriented executable model. Indeed, there is significant concurrency within the complete lifecycle, enabling different specialist groups to work in parallel, but always in the context of a homogeneous model of the system.

We accept that the HASoC methodology has yet to prove its worth by demonstrating its perceived strengths in the development of embedded systems. We firmly believe, however, that we have identified a realistic and feasible approach to the development of such systems implemented as SoC devices. We intend to continue with this work in order to hone the HASoC methodology into a useful design tool.

6: References

[1] Industry Association. *International Technology Roadmap for Semiconductors: 1999 Edition*. Austin, Texas: International Sematech, 1999.

[2] A. Ferrari and A. Sangiovanni-Vincentelli. System Design: Traditional Concepts and New Paradigms. In *Proceedings of International Conference on Computer Design*, October 1999.

[3] J. M. Rabaey, M. Potkonjak, F. Koushanfar, S-F. Li, and T. Tuan, T. Challenges and Opportunities in Broadband and Wireless Communication Designs. In *Proceedings of International Conference on Computer Aided Design*, November, 2000.

[4] F. Vahid and T. Givargis. Platform Tuning for Embedded Systems Design. *Computer*, 34(3), pp. 112-114, March, 2001.

[5] D. Morris, D. G. Evans, P. N. Green, and C. J. Theaker. *Object-Oriented Computer Systems Engineering*. Springer-Verlag, 1996.

[6] G. Booch. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[7] G. Booch. *Object-Oriented Design. Benjamin/Cummings*. 1991.

[8] D. J. Hatley and I. A. Pribhai. *Strategies for Real-Time System Specification*. Dorset House, 1988.

[9] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[10] W. Wolf. Object Oriented Co-Specification for Embedded Systems. *Microprocessors and Microsystems*, 1(20), 1996.

[11] P. N. Green, P. Rushton, and S .R. Beggs. An Example of Applying the Codesign Method MOOSE. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, IEEE Press, 1994.

[12] P. Ward, and S. Mellor. *Structured Development for Real-Time Systems*. Prentice-Hall, 1985.

[13] Object Management Group: http://www.omg.org/uml/ (December 2001).

[14] B. Selic and J. Rumbaugh. *Using UML for Modelling Complex Real-Time Systems*. ObjecTime Limited/Rational Software White Paper, 1998.

[15] G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded UML: a merger of real-time UML and co-design. In *Proceedings of CODES'01*, May, 2001.

[16] E. A. Lee, and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5), pp. 773-801, May, 1995.

[17] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation*, 1994.

[18] S. Essa and P. N. Green *System-Level Virtual Component Interfaces (VCIs) with UML*. Embedded Systems Group Internal Report, Department of Computation, UMIST, 2001.

[19] VSI Alliance System Level Design DWG. *System-Level Design Development Working Group 1 Version 1.0 (SLD 1 1.0)*, 2000.

[20] J. Daugman. High confidence visual recognition of persons by a test of statistical independence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11), November 1993.

[21] P. Branton, A. Hayes, R. Patel, S. Totten, and A. Xydeas. *HAWK: Human Access Without Keys*. Fourth Year Team Project Final Report, Department of Computation, UMIST, June, 2001.

[22] I. Jacobson, G. Booch, and Rumbaugh, J. *The Unified Software Development Process*. Addison-Wesley, 1999.

[23] S. Hebbes. Developing Embedded System Component Models from Use Cases. In *Business and Work in the Information Society: New Technologies and Applications*, IOS Press, 1999.

[24] B. P. Douglass. *Doing Hard Time*. Addison-Wesley, 1999.

[25] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavangno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach*. Kluwer, 1997.

[26] J-Y Brunel, W. M. Kruijtzer, H. J. H. N. Kenter, F. Petrot, L. Pasquier, E. A. de Kock, and W. J. M. Smits. COSY Communication IP's. In *Proceedings of the Design Automation Conference*, June 2000.