Interfacing a Simulation Engine to an Embedded Runtime Environment

B. Eames, S. Neema, T. Bapty, and J. Scott
Vanderbilt University / Institute for Software Integrated Systems
Nashville, TN, 37203

Abstract

The design of modern high-performance embedded systems is challenging. Power and size constraints limit hardware size, while performance requirements demand algorithm-specific architectures. A model-integrated approach can be used in the design capture and synthesis of these systems. A domain-specific graphical system design environment allows the capture of system requirements, design information and alternatives, and of available processing resources in the form of *models*. A model interpretation process generates architecture specifications and compilable code.

A typical first step in designing complex systems is to develop a simulation-based prototype. After functional verification of the prototype, system components are implemented, each tailored to a particular target platform. Only after all components are implemented can system integration be addressed, often uncovering inconsistencies between components and forcing costly redesign.

This paper describes an extension of a model-integrated design environment and runtime system. Simulation-based components are included in system models. A simulation engine is interfaced to a runtime environment, allowing simulation components to run "in the loop" with non-simulation based components. The extended environment allows prototype synthesis from system models and automates the integration of implemented components into the system. Further, the extended environment provides the designer with a powerful visualization and debugging tool.

1 Introduction

Ongoing research at Vanderbilt University has produced a system design tool embodying the model-integrated approach to designing complex embedded systems [1]. The design tool consists of a domain-specific graphical model editor, allowing the capture of system requirements, processing resources, and design information and alternatives. The environment also provides a model interpreter, a program which verifies model consistency and generates system specification and configuration information from the models. Figure 1 depicts the design flow, the steps a system designer would pass through when building a system using the model-integrated approach.

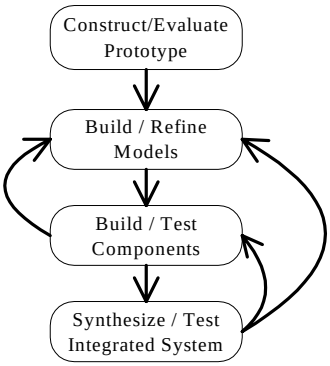

Figure 1 Design Flow used in constructing a system with the Model-Integrated Approach.

When designing systems using this approach, a first step is to construct a system prototype, exploring and developing the basic system algorithms and proving the fundamental concepts behind the system. The platform for the prototype is a simulation language and environment, such as Matlab.

After a system prototype has been constructed and tested, the system is captured in models using the domain-specific modeling environment. System algorithms are decomposed into concurrently executing processes. Inter-process communications are modeled as connections between process models. Algorithms are modeled according to the dataflow model of computation. Hardware resources and physical interconnections are also modeled.

After completing the system models, the designer must implement the processing components represented in the models. Each process model represents an implementation of a computation or algorithm tailored to a particular platform. Component implementation involves designing, implementing and testing each component individually, and, most importantly, within the context of the system.

When the components have been implemented and verified, the tools may be used to synthesize a system from the models. The synthesis operation maps processes to hardware resources, builds network initialization specifications, and generates any "glue logic" needed to facilitate inter-process and inter-processor communication. With these specifications, the network may be loaded with the synthesized system. The designer now tests the generated system and handles system integration issues.

The model-integrated approach has been shown effective in a real-world application [2]. However, there are some limitations that have become apparent through design iteration. Figure 1 depicts the design flow, including the paths of design iterations. When components are being designed and implemented, it may become apparent that a change to the system architecture is needed, requiring a refinement of the models. Also, during system-wide testing, integration may uncover inconsistencies between component implementations, requiring further changes. Each change in the models could require changes to components. Major changes to the models could result in the redesign and re-implementation of several components. The designer runs the risk of redesigning components over and over as design iterations proceed. If component inconsistencies could be discovered earlier in the design process, redesign could be avoided.

A few simple extensions of the current design tools have been made which address these limitations. A simulation engine, specifically Matlab, has been integrated into the model-integrated approach to designing systems. The modeling environment has been extended to allow the capture of the Matlab environment as a processing resource, and Matlab functions as process models. The underlying runtime system has been extended to support the execution of Matlab processes in the context of the network, and to allow data to be exchanged between Matlab functions and other processes running on the network.

2 Modeling Environment

The basis of the model-integrated approach is to model the system to be built. These models can then be used to synthesize the system.

2.1 Modeling a System

The modeling environment allows the user to capture design information about a system. The computations and algorithms used in the system are captured, as are the resources. The design tools automate the mapping of algorithms onto resources.

Resources represent any processing element used in the final platform of the system. The possible processing elements include PCs, DSPs, FPGAs and ASICs. Physical communication paths between processing elements are captured as connections between the ports of each element. Each communication path has associated with it a protocol for communication. Protocols are implemented to manage the passing of messages from one type of element to another. The use of protocols abstracts the differences between how nodes in the network send and receive data.

Algorithms used in the system are modeled as well. Algorithms consist of concurrently executing processes. Each process represents a basic block of code or logic. Inter-process communications are modeled as connections between process models. A connection represents a stream or queue of messages sent from one process to another. Each process model is assigned resource category, corresponding to the type of resource on which it may execute. Alternative implementations of a particular algorithm or process may be explicitly included in the models as well. An alternative model includes several other models, one of which will be used to implement the process. By explicitly modeling design alternatives, the modeler defines a design space, or a representation of many possible system implementations.

The design tools automate many aspects of building the actual system. A design space exploration tool aids the modeler in selecting a particular design from the design space. A code synthesis tool generates a network configuration from the models, allowing a system design to be loaded onto the network. Further, it maps a set of algorithm models onto the processing network models, associating inter-process communication streams with inter-processor communication channels. It generates initialization information for each node in the network and creates all "glue logic" needed to connect hardware-processes together. Outputs of the synthesis tool can be passed through COTS VHDL and C compilers to generate executable code, which can then be loaded onto the network.

2.2 Modeling Environment Extensions

The modeling environment described has been extended to support Matlab processes within a generated system. A new type of resource model was created to represent the Matlab processing environment. The Matlab resource model is treated much the same way as a PC or DSP processor: a processing element capable of executing processes. Process models are allowed to have a resource category of Matlab, meaning that the target implementation platform of a process is the Matlab execution environment. These extensions allow the designer to model processes implemented in the Matlab language as part of the system.

3 Runtime System Architecture

The design tools synthesize systems tailored for a particular runtime system architecture. The architecture has been designed to be easily configured through model-based system generation.

3.1 Basic Runtime Framework

The architecture consists of pieces of runtime support executing on each node of the network. For a general-purpose processor, this runtime support is provided in the form of a simple kernel. For a programmable logic device, the runtime support is provided through a virtual hardware kernel [3], containing communication support and bus arbitration. Figure 2 depicts the layers of support provided by the kernel on a processor in the system. There may be several processes allocated to a particular node. Processes can exchange data through streams. A stream represents a queue of messages, managed by the kernel, connecting a source process to a destination process. A process may send and receive messages through streams via an API provided by the kernel. The kernel is responsible for ensuring that messages enqueued into a stream reach the appropriate destination process. Many times, the destination process will reside on a different node than the source process. In such a case, the kernel will send the data to the appropriate node via one of its communication channels. Each node may have several communication channels connecting it to other nodes in the network. The kernel drives the transfer of data across a channel through software which implements a particular communication protocol.

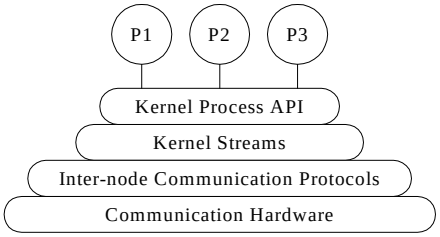

Figure 2 The kernel layers used to support inter-processor communication.

The communication protocol which drives the hardware consists of two functions, a send and a receive. The send function is responsible for retrieving a message from a kernel stream and invoking the communication hardware to send it to the connected node. The receive function queries the communication hardware to determine whether there is a message which has been sent from the connected node, and if so, transfers it to the appropriate kernel stream.

3.2 Extended Runtime Framework

The runtime framework has been extended to support the addition of a Matlab resource. Matlab provides an API through which a stand-alone program can access its processing capabilities [4]. This API is referred to as the Matlab Engine, and the runtime system is interfaced to the Matlab environment through the engine. The API allows the execution of Matlab commands, as well as the transfer of data to and from the Matlab workspace.

Figure 3 depicts the interfacing of the runtime system to the Matlab Engine.

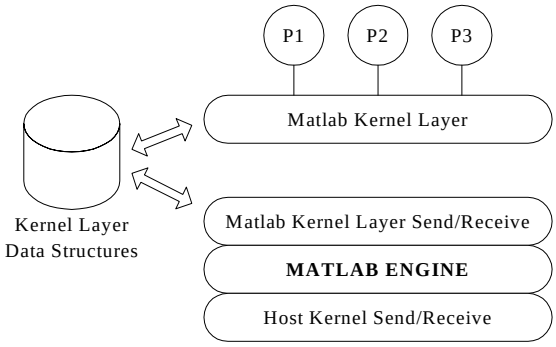

Figure 3 Interface between the host kernel and the Matlab environment.

To implement the interface between the host kernel and the Matlab environment, a new communication protocol was implemented, using the Matlab Engine software instead of physical communication hardware to perform data transfers. In order to support the execution of processes in the Matlab environment as configured and specified in the models, a software kernel layer was implemented using the Matlab language. The Matlab kernel layer is responsible for managing system message broadcast from the host and streams for inter-process communication, just as a kernel on a typical processor in the network. However, a principal difference between a typical network processor and the Matlab resource is that the Matlab environment will not execute concurrently with the host kernel. The host kernel will invoke the Matlab kernel layer through the communication protocol layer. This implementation does not physically implement the modeled semantics of the Matlab resource being a separate processing element, however, from the modeling perspective, the same behavior is achieved when execution speed is not considered (a valid assumption, considering process execution time on Matlab vs. execution time on an embedded processor).

The kernel layer in Matlab maintains system state through a set of persistent data structures, as shown in Figure 3. The communication protocol layer on the host interacts with the communication protocol layer in the Matlab environment through the engine API. Messages sent from the host are copied into arrays, which are placed into the Matlab workspace. The host layer then invokes the Matlab layer receive function, which decodes the message and places it in the appropriate Matlab stream data structure. In a similar fashion, the host receives messages from the Matlab environment by invoking the Matlab layer send function, which determines if there is a message waiting to be sent

from a Matlab process. If a valid message is ready to be sent, the message is copied into the host layer. The host layer dispatches received messages to their appropriate stream structures in the host kernel.

Whenever the host protocol layer functions are invoked, regardless of whether messages were actually transferred, the Matlab kernel layer is invoked through its entry point. The Matlab kernel layer has a single entry point, through which it invokes the system message management facilities and then the process management facilities. Process management attempts to schedule a process for execution. When a process is scheduled, it accesses streams through the kernel layer API, just as processes on a typical processor do through their kernel API. The kernel layer API functions access the data structures containing stream messages.

Through this interface to the host kernel, the Matlab execution environment can be used to perform computations and exchange data with the processing network. The communication protocol layers implemented on the host and in Matlab abstract the details of how messages can be exchanged between processes executing on the Matlab resource and the network. The Matlab kernel layer provides the execution semantics for a Matlab process which is identical to the semantics of a process running elsewhere in the network. The interface allows the Matlab environment to be seen, from the perspectives of the modeler and component builder, as just another processing element in a heterogeneous processing network.

4 A Revised Design Flow

With the extensions to the design environment and runtime system described, an improved design flow can be achieved. Figure 4 depicts the emergent design flow using the extended tools.

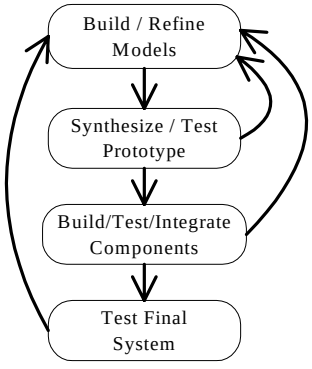

Figure 4 Design Flow used in constructing a system with the extended Model-Integrated Approach.

System construction begins with modeling the system. As a first step, a designer need not be concerned with the implementation details of the target platform on which the final system will run. Resource models may consist at this point solely of the PC host and the Matlab resource, and will be refined later in the design process. A process model representing a Matlab realization is included as an alternative implementation for each component. As each Matlab process model is created, a simulation component for the process is implemented in the Matlab language. Matlab components make use of the kernel layer API to exchange data with other processes through streams. A Matlab component can typically be implemented much quicker than a VHDL hardware component due to the versatility and power of the Matlab language and built-in functionality.

When the system algorithms have been modeled and the Matlab components have been implemented, the synthesis tools provided with the design environment can be used to generate a functional system. This system will have all of its components implemented in Matlab, and represents the system prototype. This system prototype may be loaded onto the host PC and tested. Instead of requiring the system prototype to be hand-coded, the system can be generated from a model. The benefits of modeling and system synthesis can be applied at a much earlier stage in the design process. In addition, the architecture of the prototype reflects the implementation of the final system.

After synthesizing the system prototype, the designer can test the generated system and refine the design, trying different algorithms, bit-widths, etc. When a refinement to a component interface is required, the component model and corresponding Matlab implementation are updated, and the system is quickly re-generated, and testing commences again. In this fashion, the designer may iterate, adjusting the system as needed, regenerating a prototype which is consistent with the system models. This rapid, automated prototype generation was not possible before the extensions to the tools.

When the designer is satisfied with the prototype system, component implementation can now proceed. Previously, as components were built and tested individually, each was tested with simulated inputs and a simulated processing framework. The framework simulation had to be tailored specifically for each component, and testing of a component became tedious. With the updated tools, a component can be integrated into the prototype, and the prototype system can serve as the testing and simulation framework. When implementing a component, a designer would include in the alternative model for that particular component another process model, which represents the target implementation of the component. The resource models may need to be updated to reflect the target platform of the newly included component. After

implementing the particular component, the designer then synthesizes a system with the newly implemented component replacing the Matlab version. Because the tools automate this swapping of components, the synthesis of the new system is rapid. Testing of the generated system will now focus on the newly implemented component, because all other components of the system have been verified previously. The visualization power of Matlab is very useful in this situation, allowing the designer to view, manipulate, store or inject data entering or leaving the component under test through the remaining simulation-based components in the system.

System components can be implemented in this manner, one by one, testing each in the context of the final application, using the system prototype as the testing framework. This approach saves the time of having to build a testing framework for each component, and is arguably a better means of testing components, because it tests each component in a context which is much closer to the actual execution environment. It is also possible that component implementation may uncover design inconsistencies, requiring an adjustment to the models. When this occurs, the models can be updated, along with the Matlab versions of the affected components and a prototype can be re-synthesized and tested in a controlled, step-by-step manner.

Another benefit provided by the extended tools is a better system integration. Previously, system integration could not be addressed until the components were implemented. Using the extended tools, components can be integrated into the prototype system one at a time. As each component is implemented and verified, a system using any combination of previously implemented and verified components can be synthesized and tested. This allows system integration issues to be viewed during the component-implementation phase. Prior to the tool, system integration could be a costly and time-consuming process due to design iteration caused by the need to redesign components. With the extended tools, system integration issues can be examined much earlier in the design process, at component implementation time. When system integration issues are identified and addressed at this early stage, costly design iterations, which would occur later in the design phase, are avoided. With the extended tools, design iteration is no longer penalized, but rather supported.

After all the components of the system have been implemented and verified, a final system can be synthesized from the models. This final model is then tested to be sure no system integration issues remain.

5 Conclusions

The design of high-performance, complex embedded systems is difficult. A system design tool has been shown to automate the design and implementation of such systems. Through some simple extensions, this design tool has been greatly improved. By interfacing the Matlab simulation engine to the runtime system, a better model-integrated approach to building systems has been derived. System prototypes may be synthesized from the system models. Components can now be built and tested from within the framework and context of the final system. System integration issues can now be examined as components are integrated one-by-one into the system. A new model-integrated design approach to generating complex embedded systems results, providing a solid design process and framework in which to construct systems.

Acknowledgements

This project is a DARPA Adaptive Computing Systems funded effort, involving close cooperation with US ARMY/AMICOM.

References

[1] T. Bapty, S. Neema, J. Scott, J Sztipanovits, S. Asaad, “Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems,” *VLSI Design*, Vol. 10, pp. 281-306, 2000.

[2] J. Nichols and S. Neema, "Dynamically Reconfigurable Embedded Image Processing System," Proceedings of the International Conference on Signal Processing Applications and Technology, Orlando, FL, November, 1999.

[3] J. Scott, S. Neema, T. Bapty. "Runtime Environment for Dynamically Reconfigurable Embedded Systems," Proceedings of the International Conference on Signal Processing Applications and Technology, CD-ROM Reference, Orlando, FL, November, 1999.

[4] *Matlab Application Program Interface Guide*. The MathWorks, Inc, 1998.