INTEGRATING HIGH-LEVEL SIMULATION INTO A MODEL-INTEGRATED EMBEDDED SYSTEM DESIGN TOOLSET

By

Brandon Kerry Eames

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2001

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

To Natalie and Karissa

ACKNOWLEDGEMENTS

This research was sponsored by the Defense Advanced Research Projects Agency, Information Technology Office, through the Adaptive Computing Systems project, under contract number # **DABT63-97-C-0020**.

I am grateful to those who have helped me along the way in this research. I would like to thank my research advisor Dr. Ted Bapty for providing me with the vision to understand not only the project, but also its relevance to system design. His patience in allowing me to learn along the way is greatly appreciated. I thank Dr. Gabor Karsai whose guidance through the writing of this thesis has been invaluable. Thanks to Dr. Sandeep Neema, whose technical advice and good humor helped me to persevere through the project.

I am indebted to several people throughout my education. Thanks to Dr. Ben Abbott for introducing me to in embedded systems while at Utah State University, and for helping raise my post-graduation level of expectation. Thanks to Dr. Todd Moon for forcing me to learn that hard work always pays off. Thanks to Dr. Gail Bingham for trusting me to work on interesting projects during my undergraduate career.

I thank my parents, Ollie and Janine, for their love and support. Their constant praise and encouragement helped me to set high standards, and to not settle on anything less than what I was capable.

There are not words for the gratitude I feel towards my wife, Natalie Wankier Eames. Without her support, I would not have made it to graduate school, let alone to the completion of a Master's thesis. Thanks to Karissa Rose for the sleepless nights, all the diaper changing's, but most of all, for the most adorable happy smiles.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACS – Adaptive computing systems

API – Application Program Interface

COTS – Custom Off The Shelf

FPGA – Field Programmable Gate Array

HDL – Hardware Description Language

MIC – Model Integrated Computing

VHDL – Very high speed integrated circuit Hardware Description Language

CHAPTER I

INTRODUCTION

The design and implementation of complex embedded systems is difficult. New and innovative applications push stringent requirements, necessitating improvements in technologies and system design methodologies. Current design philosophy is highly dependent on simulation. Simulation provides a means to develop, test, and evaluate designs prior to committing to implementation, allowing design flaws to be detected and corrected early in the design process. Model-based approaches to system design have been introduced to facilitate the design of complex systems at a higher level of abstraction. The integration of simulation capabilities into a model-integrated embedded system design tool provides an improved framework for developing complex embedded systems.

Embedded Systems

Embedded systems form a broad class of computer-based systems. In general, an embedded system is a computer system that interacts directly and dynamically with its environment. These interactions are often facilitated through sensors to discern the state of the environment, and actuators to change or update the state of the environment. Embedded systems are used in a wide variety of applications, from military domains to end-user products. Examples of such systems include digital cellular telephones, anti-lock braking systems in automobiles, flight control systems in avionics, and missile guidance systems. This thesis discusses the design of a specific class of embedded

systems, that of digital signal- or image- processing systems. When embedded systems are mentioned in this thesis, we refer to this particular class of embedded systems.

Traditional System Design Approach

The first step in an embedded system design is to clearly define the system to be designed. The system stakeholders hold discussions with the designers until arriving at a high-level understanding of what the system will do, as well as a general idea of how it will be constructed. System design requirements are formed from these discussions, documenting what the system will do. The high-level concept of how the system will be constructed is documented in the form of architecture diagrams and system-level block diagrams. After iterating with the stakeholders through this process of developing and analyzing requirements and high-level design architectures, the designers may proceed with the more detailed system design work, driven by the requirements.

The steps involved in the detailed system design are depicted in Figure 1. In the first step, a developer constructs a simulation of all or parts of the initial high-level system architecture, with the intent of not only visualizing the system, but of verifying the high-level design against the system requirements. Issues discovered at this stage may require the refinement of the high-level design architecture or perhaps further refinement of the system requirements. It is important to note that issues which go undiscovered at this stage will propagate into later design stages, and, when discovered can be difficult to correct. For this purpose, simulation is utilized early in the design process, to ensure the initial design is correct at a high level. Another purpose of simulation at this stage is to gather sufficient information about the system to allow rough analyses of design approaches and tradeoffs. Simulations are designed to provide

sufficient accuracy and detail to allow the detection of design flaws. However, because little is known at this stage about the lower level details of the design, it is not possible to obtain highly accurate simulations. Therefore, as a general rule, at this stage accuracy and detail are traded in favor of rapid execution. Designers often make use of a simulation language and/or package such as Matlab [13] to develop these high-level simulations. Matlab facilitates the rapid development and evaluation of simulation prototypes through its powerful data visualization capabilities and extensive libraries.

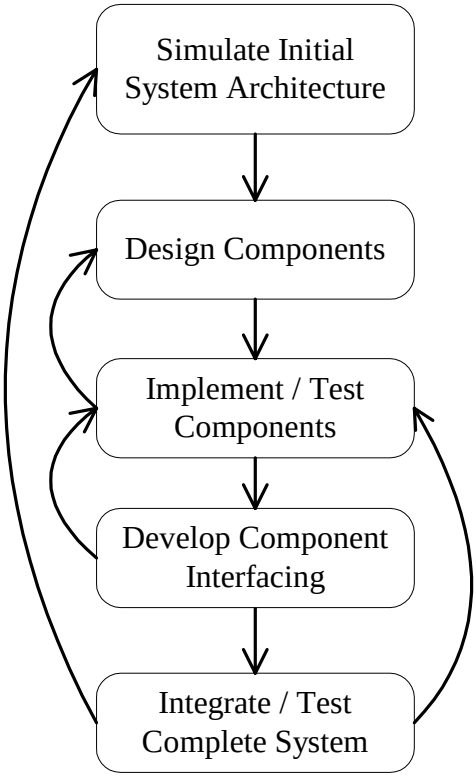

Figure 1. Design flow often used in embedded system development

After simulating the initial design architecture, the designer proceeds to design the system components. Components form the fundamental building blocks of the application. Each component is designed individually. Good design practice dictates that

components should conform to well-defined and documented interfaces, and should provide specific entry points. A first step in the development of a component is to construct a simulation of the component's behavior. The purpose of the component simulation is the same as the simulation of the initial design architecture, to verify the design against the system requirements, or to verify that the component does what it is intended to do. Only after this verification through simulation has been performed should the designer proceed to the more tedious and expensive step of component implementation. Upon completion of the component implementation, the developer tests the component by simulating the inputs and execution environment of the component. The behavior of the component implementation can be compared against the component simulation to verify proper functionality. The developer iterates over this process until all components have been implemented and verified.

The next step in the design process is system integration. At a high level, integration involves properly connecting components together to form the final system. The code used to connect components is often referred to as "glue" or "glue code" for obvious reasons. The developer must tailor glue code for the particular components in the system. Another part of system integration is the configuration of runtime middleware to properly support the execution of the system components.

At first, the concept of system integration may seem trivial. However, issues can arise at this late design stage, which force costly redesign efforts. For example, components could export incompatible or inconsistent interfaces. In this case, it is possible that the individual components were designed and implemented properly, but the system cannot be fully integrated. The result is that one or more of the affected

components will need to be refined and tested again. Situations such as these stall the design process, forcing the designer to iterate to a previous stage in the design flow. These types of design iterations are costly, especially when discovered late in the design. When finally the designer succeeds in interfacing all the components together, the system as a whole is tested against the requirements.

Simulation in System Design

As illustrated above, simulation plays an important role in system design. Simulation allows the incremental verification of a system during development, allowing flaws to be uncovered early in the design. Further, simulation allows application development to proceed simultaneously with, or even prior to, the development of the system execution platform.

There are two basic types of simulation: performance and functional. A performance simulator abstracts the details of how a system performs its tasks in favor of simulating the temporal aspects of the system. A functional simulation captures the behavior of a system, allowing the designer to verify whether a system meets functional requirements. Simulators also vary in their level of detail and accuracy. A high-level simulator offers a less detailed, less accurate view of a system in favor of a rapid execution time. A high-level simulation is used when low-level system details are not needed or are not known. A low-level simulator provides fine-grained, more accurate details at the cost of longer simulation times.

Many different simulators are in use today. For software, instruction set simulators allow software to be interpreted and “executed,” allowing a developer to trace through source code and view the internal state of a simulated processor. For hardware

components, HDL simulators allow designs to be simulated and signals to be examined. There are many different levels of HDL simulation, each offering a different level of resolution, and each requiring a different execution time. Some products claim "co-simulation" capabilities [21], where a developer provides models of hardware and software, and the simulation records their interactions on execution. There are a plethora of development tools currently on the market offering simulation capabilities. It is beyond the scope of this thesis to provide an exhaustive review of these tools. However, see [16] for an example of an instruction set simulator, and [17] for an overview of different HDL simulators and a sampling of HDL simulation vendors. Another product of interest is Simulink by The MathWorks, Inc [18]. Simulink allows a user to graphically represent a system as a set of block diagrams. Each block is driven by a Matlab function, and Simulink provides the glue to connect each block and facilitate data exchange. Simulink interfaces with a product called xPC [19], allowing a target embedded processor to execute actual system components "in-the-loop" with system simulation code. A developer may model a system using Simulink and then execute actual system components as if they were part of the simulation. This approach is similar to the virtual prototyping discussed in this thesis.

A Model-Integrated System Design Approach

Another approach to system design involves modeling. A model is an abstraction or higher-level representation of a system and its components. Through modeling a system, a designer can focus on those details which are most important and relevant at each stage in the design. A model-integrated approach to system design involves the construction of system models using a graphical domain specific modeling language.

System components and interconnections can be represented graphically as signal flow diagrams. The system execution platform can be captured as block diagrams representing processing elements, with interconnections representing communication links. A translator program, called a model interpreter, is then executed, which synthesizes an executable system from the diagrams. The translator automates system integration by generating the necessary component interfacing code, as well as the runtime middleware configurations. The designer must still design and implement the system components, but the model interpreter handles the details of system integration. The power of the model-integrated approach lies in the abstraction of unnecessary or redundant details, allowing a designer to focus on what is most important in system design. [15] discusses the general concepts behind Model-Integrated Computing (MIC).

A tool embodying the principles of model-integrated system design for embedded systems has been developed as part of the Adaptive Computing Systems project at the Institute for Software Integrated Systems [10]. This design tool allows a developer to model complex embedded signal and image processing systems, and provides system synthesis capabilities. The toolset greatly simplifies many of the complexities associated with the design of complex signal processing systems. The design tool, however, does not provide any integrated simulation capabilities, the benefits of which have been discussed. While a designer can simply use other tools to perform system simulations and then use the ACS toolset to model and synthesize a system, a tool merging the modeling capabilities of the ACS toolset with the benefits of simulation will aid the developer in many ways. The integrated toolset will allow the development of systems from a single design representation, instead of the separate representations required for

each of the various simulators used, as well as the system models captured in the toolset. Further, the integrated toolset will allow the developer to apply simulation techniques in the model-integrated design process, aiding the development of high-level simulations of the initial concept design architecture, as well as component design simulations. Integrating simulation into the ACS toolset not only integrates the benefits of system simulation in the model-integrated design approach, but also streamlines the design flow of system development.

Model-Integrated Simulation Package

This thesis discusses the integration of high-level simulation capabilities into the ACS system design toolset. Chapter II provides as background a detailed introduction to model integrated computing and the ACS system design toolset. Chapter III describes an extension to the ACS toolset, which generates high-level functional simulations directly from the signal flow diagrams and component models, facilitating simulation during model construction. Chapter IV discusses the integration of virtual prototyping into the ACS toolset, allowing pieces of the system to execute in simulation concurrently with actual component implementations at runtime. Virtual prototyping provides a framework not only for visualizing the system, but also for simulating and testing components as they are developed. Chapter V analyzes the effects of these toolset extensions on the ACS design flow, and Chapter VI discusses the thesis conclusions and future work.

CHAPTER II

BACKGROUND: MIC AND THE ACS TOOLSET

The embedded system design community has exerted much effort in creating sophisticated high-level system design tools to simplify the complexities of embedded system design. A design tool incorporating the concepts of Model-Integrated Computing has been produced as a product of the Adaptive Computing Systems (ACS) project at the Institute for Software Integrated Systems [10]. The ACS toolset facilitates the design and implementation of high-performance adaptive signal and image processing embedded systems.

Model-Integrated Approach to System Design

Model-Integrated Computing is an approach to the analysis and development of information systems. It involves the use of models to represent domain concepts at a high level of abstraction. The model-integrated approach to system design [14] involves graphically representing a system at a high level of abstraction using a domain specific graphical modeling language, and then performing system synthesis from the models. The modeling language, also called a paradigm, embodies concepts from the domain of the system to be designed. A generic modeling editor is configured to support the domain specific modeling language, allowing a user to capture system components and specifications abstractly in the form of models. Insignificant or unnecessary details about the system are omitted from the models, allowing the user to focus on those aspects of the system that are most relevant to the system design. After a system is modeled, a

translator program, called a model interpreter, is invoked to perform useful translations from the models. Precisely what translations take place depends on the application domain. For example, an interpreter could be constructed to translate information captured in the models into input for a domain-specific analysis tool. For an embedded systems domain, an interpreter could be constructed which translates system models into code and runtime configurations for an embedded system.

There are several benefits of the model-integrated approach to system design. By abstracting away unnecessary details through system modeling, the developer can focus on those aspects of the system which are most important during the design. This abstraction facilitates the design of complex systems at a much higher level, mitigating much of the complexity, allowing complex systems to be constructed correctly and efficiently. The concept of system synthesis through model interpretation is the vehicle for facilitating the abstraction. The interpreter shields the developer from many of the low-level system details which are unimportant at the system level. The model-integrated approach is to an extent flexible to design iteration and requirements change, because systems are synthesized from models. When a change in the design is required, a simple update to the models is made along with any needed updates to user-developed components, and the system is re-synthesized. This approach to system design and development has been demonstrated in several different applications [1][3][4][5][6][7][8][9].

The ACS System Design Toolset

The ACS design toolset facilitates the development of adaptive signal- and image-processing embedded systems. Signal- and image- processing systems can be

represented as signal flow diagrams, with blocks in the diagram representing system components, and connections as paths for data exchange between components. An adaptive system is defined as a system which transitions between discrete modes of execution, where each mode consists of a distinct set of components and interconnections. When an adaptive system transition from one mode to another, the set of components corresponding to the second mode is activated, replacing the set of components from the first mode. A missile guidance system can be constructed as an adaptive image processing system. In the first mode, the system seeks multiple targets at a long range, and lower frame rates and lower power consumption levels are acceptable. As the missile nears the general area of the possible targets, the system transitions to a new mode where the frame rate is increased while the system attempts to single out a small set of distinct targets, determining the best target to track. At a close range to the determined target, the missile enters a third and final mode, where the frame rate is at a maximum, and the missile focuses on a single target, tracking all changes in its position until impact. The three distinct processing modes of the system have different goals and requirements; yet together form a single system. Obviously, the design of adaptive systems is difficult, involving many issues. The ACS toolset was developed to mitigate many of the complexities behind the design of such systems.

The ACS toolset embodies the principles of the model-integrated approach to system design. It provides a rich graphical modeling language and graphical model editor to allow the capture of system characteristics and specifications as models. A model interpreter is provided, which translates system models into a set of configurations for an embedded runtime environment, which has been designed to support component

execution on a heterogeneous network of processing elements. The following sections describe the ACS graphical modeling language, the supported runtime environment, and the model interpreter.

The ACS Modeling Language

The ACS modeling paradigm allows the capture of applications at a high level of abstraction. When designing an adaptive system, the three principle areas of concern are the development of system components and their interactions, the establishment of the different modes of execution and the transition conditions governing mode transitions, and the development of the execution platform on which the application will run. The modeling paradigm supports each of these three areas of design by allowing the user to independently model the hardware or processing resources of the system, the components and their interconnections, and the adaptive behavior governing the modes and transitions.

Modeling System Resources

As adaptive systems typically execute on heterogeneous processing networks, the paradigm supports the modeling of several distinct processing elements. FPGAs, ASICs, PCs, DSPs, general-purpose processors and Memory are represented as blocks in a block diagram. Boards or cards are captured as collections of basic resources. Point-to-point communication links between resources are captured as connections between the ports of different blocks. Figure 2 shows an example of a resource model. This figure represents a heterogeneous signal processing platform, with a host processor connected to a TMS320C40 DSP, which is connected to a TMS320C67 and an Altera FPGA. The FPGA is connected to a local bank of SDRAM. Each connection represents a physical

communication link or channel between resources. As links in the network each realize a communication protocol, the modeler may select the appropriate protocol for a link by setting the Protocol attribute of the ports on either end of the link. By modeling system resources separate from the application components, details of the application development have been abstracted away from the platform development. However, sufficient information has been captured in the resource models to facilitate application development.

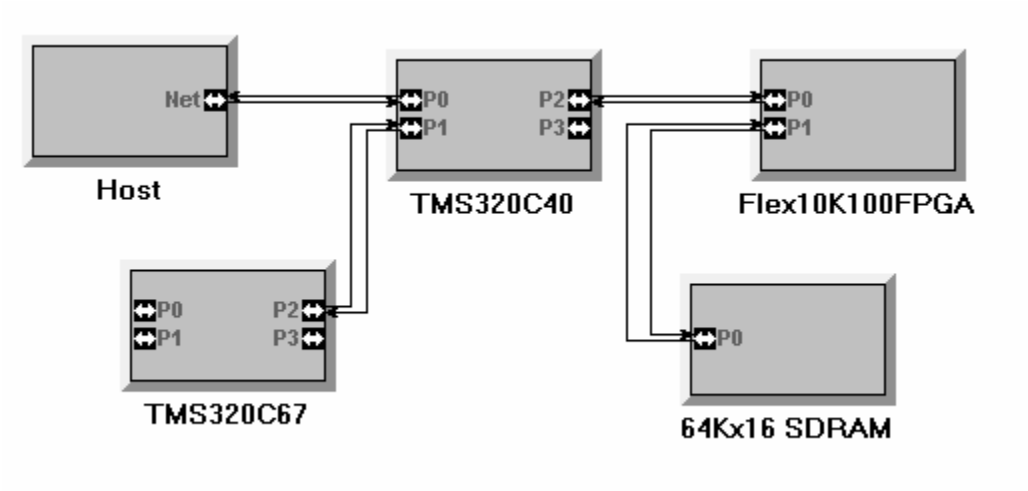

Figure 2. Model of system resources, showing a heterogeneous processing network

Modeling System Components

As previously described, adaptive systems consist of concurrently executing and communicating components. Components exchange data through message passing. Components and their interactions can be conveniently represented as a type of signal flow diagram, with the nodes of the diagram representing the components, and the connections representing streams through which messages are passed from one component to another. The paradigm supports the capture of these signal flow diagrams

according to the dataflow formalism. The paradigm also supports hierarchical composition of components, facilitating the capture of complex designs in a compact format. Atomic components (those which do not contain other components) are referred to as primitives and are captured in the modeling paradigm as ProcessingPrimitive models, while hierarchical or composed components (constructed from the components contained in it) are referred to as compounds and are captured as ProcessingCompound models. Primitives are realized by the developer as code or VHDL hardware.

The modeling paradigm also supports the capture of design alternatives. A model called a ProcessingTemplate or template can be included to represent a collection of alternatives. At model building time, the developer is allowed to specify several alternative implementations for a given component by including a model of each in a template. In a later design stage, the model interpreter will allow the developer to select which of the alternatives is to be actually used in the final system. The selected component contained in the template will effectively replace the template in the model hierarchy, and the remaining alternatives will be ignored. By explicitly including design alternatives in the models, a developer is not forced into a single implementation early in the design phase.

Component Composition Through Dataflow

Component interactions are captured following the semantics of the dataflow formalism. A compound component is captured as a signal flow block diagram. The blocks represent other components, and the directed connections between the blocks represent directed channels through which messages are passed. Data is said to flow through the network during execution because when a component executes, it consumes

inputs and produces outputs. Inputs are removed from the message channels connected to the inputs of a component, while outputs are enqueued into the message channels connected to the outputs of the component. Typically, a component can only execute when some or all of its input channels have messages available for consumption. Thus the order of execution of the components is a function of the state of the message channels, or the data flowing through the network.

There are two general classes of dataflow models, asynchronous and synchronous. In synchronous dataflow [20], sufficient information is captured in the models to be able to schedule the order of component execution at model interpretation time. The necessary information to determine schedulability is the number of messages or tokens a component will consume on each input channel on each invocation, as well as the number of tokens on each output the component will produce per invocation. If this information is known at model-building time, a schedule of execution for the components can be constructed [20]. In many systems, however, it is not known at model building time how many tokens each component will consume and produce, and some components may require a variable number of tokens per invocation. For systems containing such components, asynchronous dataflow can be used to represent the system. Component scheduling in an asynchronous dataflow system is performed strictly at runtime. In the runtime environment provided with the ACS toolset, the components participate in determining their own schedulability. The runtime environment maintains the channels connecting components, and allows components to access the channels through an API. When a component is invoked by the runtime environment, it is responsible for determining whether there are sufficient input tokens awaiting

consumption to allow the component to execute. Further, because all message channels have finite buffer space, the component must determine if there is sufficient space in the channels connected to its outputs to hold the tokens to be produced by its execution. If both conditions are met, the component performs its computation. If not, the component yields control to the runtime environment. Scheduling proceeds in this cooperative manner, with the runtime environment simply executing all components in a round-robin fashion, and those that determine themselves "ready" to execute, perform their computation, while those that do not, wait for a future invocation.

The modeling paradigm supports the semantics of asynchronous dataflow for representing components and component interactions. Each component can contain input and output ports. A port can be connected to a port of another component. Such a connection models a channel through which the first component can send messages to the second.

Hierarchy in composed models is merely a means of visually simplifying a signal flow diagram. Because complex adaptive systems often contain several components, a diagram representing all the primitive components in one level would be very difficult to comprehend. Compound components were introduced into the modeling language to facilitate the capture of systems with several components, through allowing complex components to be represented as compositions of simpler components. However, at runtime, only the primitive components will actually form part of the system. The model interpreter flattens the hierarchy by replacing each compound component with the components and interconnections of which it is composed.

A simple control system application can be developed using the ACS toolset. For example, consider a system consisting of two sensors that detect information about the physical state of the system to be controlled, an actuator which adjusts the state of the system, and a software controller which reads the sensors, calculates any required adjustments to the state of the system, and sends control signals to the actuators. A model of the system controller is represented in Figure 3. The two leftmost components, ReadSensorA and ReadSensorB are ProcessingPrimitive models, and represent components responsible for reading the sensor information. The comparison component is represented as a ProcessingTemplate model, containing models of alternative implementations of the comparison algorithm. The GenerateCorrection block is a compound, representing group of components whose aggregate behavior implement the correction generation. One of the components within the GenerateCorrection compound is responsible for interacting with the system actuators (not shown in the figure). All solid connections in the figure represent dataflow connections. The purpose of the InitialStatus icon will become clear in a later section.

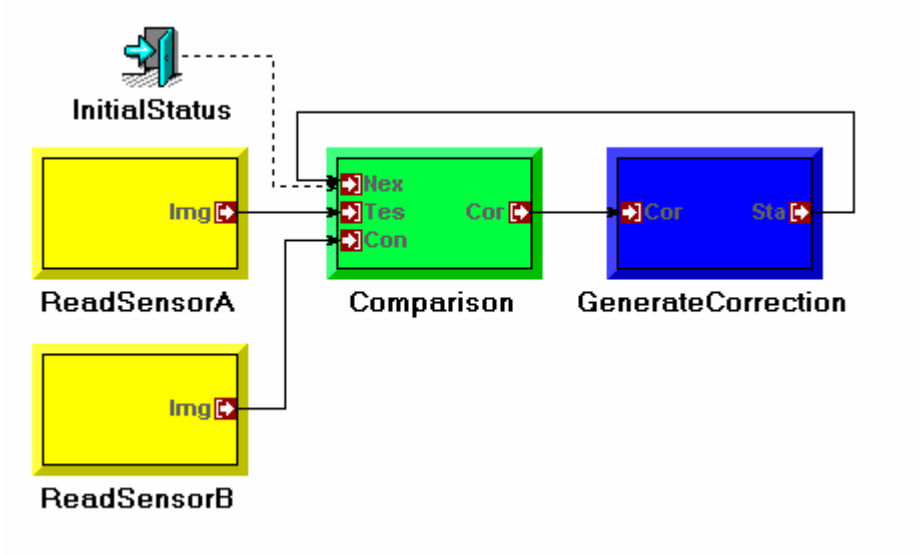

Figure 3. Example of ProcessingPrimitive, ProcessingTemplate, and ProcessingCompound models.

Modeling System Modes and Mode Transitions

As stated previously, the paradigm allows the modeler to represent the adaptive behavior of the system. System modes are captured as the states of a finite state machine. Transitions between modes are captured as the conditions governing the transition between the states. Transitions are conditioned on global system events, which are also represented. Each mode is associated with a top-level compound model, representing the computations the system is to perform while in that mode of execution.

ACS Runtime Environment

The ACS toolset provides a runtime environment tailored for seamless integration into the model-integrated approach to system design [11]. At runtime, adaptive systems consist of multiple components executing concurrently on multiple processing resources. The runtime environment provides services and support for system components, abstracting from the components the details of inter-component and inter-node

communication. The runtime environment consists of a standalone kernel for each processing node in the network. For FPGA nodes, the kernel is a virtual hardware kernel, providing for bus arbitration and communication port sharing. For processor nodes, the runtime environment consists of a thin real-time kernel which manages inter-process communication, as well as a deterministic dynamic memory management layer. The real-time kernel for a given processor is responsible for scheduling all processes mapped to that processor. It also facilitates the broadcasting and dispatching of kernel-level messages across the network as they are received from other nodes or from the host. The runtime environment supports dynamic reconfiguration by allowing the host to broadcast code corresponding to a new mode through the network followed by a reconfiguration command. The network will then reinitialize itself and begin executing the new mode.

ACS Model Interpreter

The ACS model interpreter is responsible for mapping system modes to runtime system configurations. The model interpreter automates the process of selecting between design alternatives, allowing the developer to select a single point design from the design space. Next, the interpreter generates code to configure the runtime environment to support the execution of the components captured in the models. The developer must select a mapping between components and processing resources, either by explicit referencing between models in the modeling environment or by selecting one of a set of mappings offered by the model interpretation process. The interpreter generates configuration code for each of the resources in the network to allow each resource to support the components that are mapped to it. For FPGA components, the interpreter generates VHDL code to connect the primitive components to the virtual hardware

kernel. The output of the model interpreter may then be compiled by COTS tools, specific to the particular resources used in the network, and then loaded and run.

Because systems are synthesized in this fashion from the models, many issues in the non-model-based approach to system design are avoided. For example, primitive component implementations must conform to the interface supported by the runtime environment. By specifying the interface in the models, the interpreter may then generate a configuration of the runtime environment to support that particular interface. When system integration is performed, the only possible interfacing inconsistency that can occur is if the developer incorrectly specifies or implements a component. System synthesis generates the integrated system, therefore a system will be correct by construction, assuming the model interpreter properly performs its function and the primitive components are correct.

ACS System Design Flow

The ACS toolset provides a framework for developing adaptive embedded systems. After the stakeholders and designers have established the system requirements and have agreed on a high-level system architecture, the detailed design using the ACS toolset may begin. The steps taken in system development when using this toolset are depicted in Figure 4. The first step is to model the system using the graphical modeling language. During this step, the developer begins with the high-level system architecture, and recursively refines complex components into simpler components until arriving at the primitive level. Paths of data exchanges between components are captured as well. The different modes of operation are derived, along with the conditions for mode transitioning. The resources of the system are captured in the resource models, as are the

communication links between resources. After this step, the developer proceeds to implement and test individual primitive system components. Primitives implement the interface captured in the models, and after implementation, should be thoroughly tested to ensure proper behavior. After implementing the components, the full system can be synthesized through model interpretation and tested. It should be noted that even though the system is synthesized from the models and the components, and each component has been individually tested, the developer still needs to test the integrated system to ensure complete consistency.

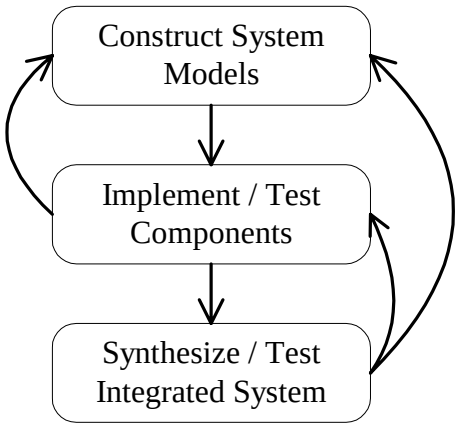

Figure 4. Design flow applied when using the ACS system design tools to construct embedded systems.

Iteration is an inevitable part of any system design. Iteration occurs when a problem or issue is uncovered in a later design phase, causing the design process to start again from a higher level and proceed again. Obviously, excessive iteration is wasteful. However, iteration must be expected in system design, because early in the design process developers are not always aware of all issues pertaining to the system under construction, so many issues must be dealt with as they are discovered. Also, even the best developers are not infallible, and mistakes, however minor, will be made. The ACS

design flow provides flexible paths for design iteration to occur. Due to the intuitive graphical nature of the modeling environment, constructing and updating models is not difficult. When component implementation uncovers, for example, a problem with the modeled interface to a component, it is not difficult to quickly change the modeled interface and proceed with the implementation. When system integration uncovers a problem with a component implementation, that component will simply be adjusted and system testing may proceed. If system integration uncovers a problem requiring an adjustment to the models, it is a simple issue to update the models, update the corresponding component interfaces, and re-synthesize the system. The ACS toolset offers a flexible and intuitive infrastructure for developing embedded systems.

However, the ACS toolset supports no concept of simulation, the benefits of which were discussed in Chapter I. Even though the developer may perform simulations in separate tools, an integrated support for simulation will allow system development to proceed from a single design representation. Further, as a consequence of integrating simulation capabilities with the ACS runtime environment, an infrastructure for component testing and debugging is developed.

CHAPTER III

HIGH-LEVEL FUNCTIONAL SIMULATION SYNTHESIS

When modeling a system, a designer may want to simulate a design prior to committing to implementation. A functional simulation is a representation of a system which can be executed to verify intended system behavior. A high level functional simulation allows a designer to check a coarse-grained design against general functional system requirements. This type of simulation is often performed when laying out a block-level representation of a system, and allows a developer to check for errors and inconsistencies in the design before proceeding to a lower-level, more detailed stage of development. The MatSim model interpreter has been developed to generate a functional simulation from a set of application models. This simulation can then be used to compare modeled behavior against high-level functional requirements.

As stated in Chapter I, a first step in the detailed design of a system is to simulate the initial design architecture derived from the requirements analysis design phase. Chapter I also discussed the popularity of the Matlab language and environment as a platform on which to develop these high-level simulations. A developer can easily model the initial design architecture using the ACS modeling language and environment. MatSim provides the ability to translate these initial system models, subject to a certain set of constraints on modeling semantics, into a set of Matlab subroutines which can be executed together with user-provided primitive component simulation subroutines to simulate the system.

The goal of the MatSim tool is to allow the developer to simulate complex components. As discussed in Chapter II, complex components are known as compound components, and are composed of primitive components and other compound components. It is not the goal of MatSim to simulate the adaptivity of an adaptive system. Generated simulations represent a single mode of operation. However, the developer may use MatSim to generate simulations for each individual system mode. MatSim does not attempt to simulate component executions on the individual processing resources. MatSim generates a high-level functional simulation for a modeled system. The details of executing a component on one resource versus another are abstracted away at this level. Further, MatSim does not attempt to generate a simulation for a design space. As such, it is assumed that all design alternatives have been resolved through the ACS model interpreter, and that the input to the MatSim tool is a single point design from the modeled design space.

In keeping with the philosophy of the ACS toolset that the precise semantics of primitive components are not captured in the toolset, MatSim does not attempt to generate simulation code for the primitive components of a system. Instead, MatSim assumed that the developer provides a Matlab function to simulate each primitive component in the system. MatSim generates a Matlab code framework to invoke the primitive simulation functions provided by the developer. A drawback to this approach, as shall be seen, is that in the development of MatSim, it was necessary to make a simplifying assumption about the component diagrams, affecting the semantics of the system models. MatSim assumes that system models conform to a restricted set of dataflow semantics. MatSim will not generate a correct simulation of components which

do not conform to the restricted semantics. Details of the semantic restrictions will be enumerated in a later section.

MatSim is meant to be a tool to be used during the model construction stage of system design. During the system modeling stage, a developer recursively breaks complex components into simpler pieces until representing the simple atomic components as primitives. When modeling the initial design architecture, the designer can represent complex components as primitives to achieve an initial design model. A simulation for each of the complex primitives can then implemented in the Matlab language, and MatSim can be used to synthesize a simulation for the design. The developer can then verify the initial design concept against high-level functional requirements. The designer next proceeds to break complex components into simpler pieces by replacing primitive models representing complex components by compound models containing simpler components. This design can be simulated by breaking the code representing complex primitives into code representing simpler primitives. The developer can invoke MatSim again to generate the simulation framework around the new set of component simulations. At each step of the design, the developer may generate a simulation of the modeled system (assuming the restricted modeling semantics required by MatSim have been applied) by creating simulations for each primitive component in the system and then invoking MatSim to create the framework. The system modeling design stage is completed when all components have been resolved into their level of granularity as appropriate for the design. Following the process of breaking complex component simulations into several simpler component simulations according to the recursive decomposition represented in the models provides the developer at the end

of system modeling with a high-level Matlab simulation of each component in the system. Assuming the behavior resulting from the MatSim-generated simulations has been verified by the developer, at the end of the modeling design stage, the final Matlab component simulations can be used as functional specifications for component development in the next design stage.

The MatSim tool allows the developer to generate Matlab simulations directly from component models. The following sections describe the semantics of this translation, along with its corresponding modeling implications.

Matlab Representation of System Component Models

As discussed in Chapter II, components are modeled in the form of hierarchical signal flow diagrams. Primitive system components are represented in the ACS modeling language as ProcessingPrimitive models, while collections of components are represented as ProcessingCompound models. The next sections discuss how MatSim generates a functional simulation from these types of models.

Primitive Component Models

Primitive components form the basic building blocks of the system. In Matlab, the basic block of code is a function or subroutine, and as such, primitive components are represented as Matlab functions. As stated previously, the behavior of primitive components is not captured in the ACS modeling tool, so the developer is required to supply the Matlab functions simulating the intended behavior of each system primitive. MatSim generates a framework around these user-provided primitive functions. The generated framework is responsible, among other things, for ensuring the proper

invocation of each component. Thus, MatSim must have the ability to discern how to invoke the function based on the model of the function. In order for the framework generated by MatSim to properly invoke a user-provided primitive function, the function signature or prototype must match the port signature of the primitive model it represents. MatSim assumes that the ports of a primitive model represent parameters or arguments of the model's corresponding function. Each input port represents an input parameter, and each output port an output parameter. The name MatSim uses to generate the function invocation is derived from the "Script/Component name" attribute of the ProcessingPrimitive model. Figure 5 depicts a ProcessingPrimitive model named DoCorrection. The model has two input ports, Correction and Position, and output port Out. The "Script/Component name" attribute has been set to "doCorrection." MatSim assumes a correspondence between the name of the function realizing the primitive and the scriptname attribute. MatSim requires the primitive simulation function to have the letter "M" concatenated with the name provided in the scriptname attribute as its name. The addition of the "M" to the beginning of the name avoids certain name mangling issues which arise due to system modifications discussed in Chapter IV. The function that simulates the DoCorrection component should therefore be named "MdoCorrection". The function MdoCorrection, shown in Figure 6, correctly represents the DoCorrection model of Figure 5.

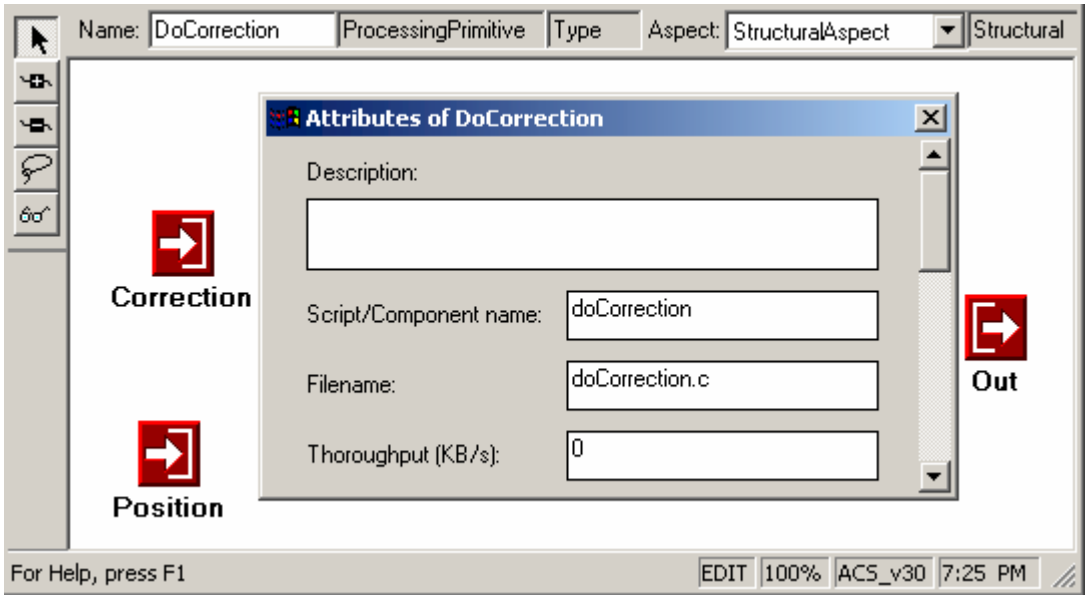

Figure 5. A ProcessingPrimitive model with ports and a scriptname, to be translated into a Matlab function

```
function [Out] = MdoCorrection(Correction, Position)
%function doCorrection to simulate the correction
% runtime component.

%P-constant set to 0.375 by experimentation
P_constant = 0.375;
Out = Position * Correction * P_constant;
```

Figure 6. User-provided function represented by the Primitive model in Figure 5.

If the framework generated by MatSim is to properly invoke the user-provided primitive functions, parameter passing is significant. Not only must the correct number of parameters be passed, but those parameters must be passed in the proper order. The proper ordering of parameters is determined by the numbering assigned to the ports in the models. When building models, the designer assigns a number to each input port and each output port by setting the corresponding port number attribute field. Ports of a given model should be numbered sequentially, starting with zero, and input ports are numbered independent of output ports. In the case of the DoCorrection model of Figure 5, the

Correction port's port number attribute has been set to 0, while that of the Position port has been set to 1. The output port Out's port number has been set to 1. The code in Figure 6 correctly corresponds to the DoCorrection primitive model, having the Correction parameter first in the input parameter list, followed by Position, and Out as the only output parameter. When the developer follows the conventions stated here about parameter ordering, as well as function naming, the framework generated by MatSim will properly invoke the user-supplied simulation components.

Compound Component Models

A ProcessingCompound model represents a collection of components. As each primitive component is represented in Matlab as a Matlab function, a collection of components can itself be represented as a Matlab function. This convention leads to a simple representation of a system in Matlab. A compound simply represents a function, whose contents are calls to other functions. Those functions represent the components contained in the compound. The ports of a compound represent the parameters of the corresponding function. The code representing each compound in a set of models can be generated by the MatSim model interpreter, and the user simply needs to execute the function representing the top level model in the hierarchy to execute the simulation. There are, however, several issues which arise with this representation, not the least of which is the semantics of the dataflow representation of the components. Another issue is the fact that Matlab is a sequential programming language, only allowing one function call to be active at a time. The systems modeled in the ACS toolset consist of concurrently executing processes. In order for the generated Matlab simulation to truly represent the semantics of the modeled system, the functions representing the system

components would be required to execute and exchange data concurrently. These deficiencies in the MatSim simulation semantics are addressed in the following sections.

Dataflow and Static Scheduling

As discussed in Chapter II, regarding the ACS toolset, the scheduling of primitive components follows the execution semantics of asynchronous dataflow. This means that the schedule of component execution is not known until runtime. In the context of the ACS toolset, a component participates in its own scheduling by determining whether the proper conditions have been met to allow the component to execute. However, the basic concept behind a component is still present, in that, when it executes, it consumes input and produces output, and the schedule of invocation is a function of the state of the input and output buffers attached to the component. Exactly how many tokens a component consumes on each input and produces for each output is not captured in a diagram following asynchronous dataflow semantics.

The Matlab language does not support asynchronous dataflow execution semantics. The language provides an explicit control flow structure. Assuming a Matlab function consists of a series of function calls, as a consequence of the explicit structure imposed by the Matlab language, the order in which the function calls should be made must be known when the containing function is written. This ordering cannot vary when the function executes. A convention could be implemented which mimics the dynamic scheduling of the ACS runtime environment, allowing the components themselves to participate in their scheduling. However, a goal of the MatSim tool is to allow the developer the freedom to generate simple Matlab functions representing the primitive components of the system, without being burdened with the necessity of interacting with

a simulation runtime environment or API, as required by the ACS runtime environment. To avoid the need for dynamic scheduling, the MatSim tool is required to generate a schedule for properly invoking the functions called by a compound function. There is insufficient information available in an asynchronous dataflow graph to determine a correct static schedule, so MatSim was designed to make an assumption about the semantics of the dataflow graph represented in the models.

The synchronous dataflow formalism was discussed in Chapter II. MatSim schedules function invocations assuming the component models were created following the synchronous dataflow formalism, with each input port requiring a single token, and each output port producing a single token on each invocation. Only systems that implement these semantics can be properly simulated using a simulation generated by the MatSim interpreter. However, systems that do not exhibit these semantics can emulate this behavior through modifications to the simulation code, by allowing components to maintain state, effectively buffering inputs and outputs as required.

In order to produce the correct static schedule for a compound model, MatSim must resolve the data dependencies between the components contained in the compound. These data dependencies can be conveniently represented as a directed graph. Nodes of the graph represent the components contained in the compound. Edges in the graph represent a dataflow path connecting models. If two models have multiple paths connecting them (in the same direction), a single directed edge is sufficient to model the dependency. It is assumed that data is always present at the input ports and data can always be written to the output ports of the parent compound. Connections between a port of a contained model and a port of the container are not represented in the digraph,

because no dependency exists. (This assumption is valid because the ports of a model represent parameters of a function, and the parameters of a function persist across all function calls made within that function, and can be read from and written to at any time.) The resulting directed graph represents the data dependencies between the components contained in a compound. For example, Figure 7 depicts a compound model, GenCorrection, containing two components, AcquirePosition, and DoCorrection. The graph representing the data dependencies is depicted in Figure 8.

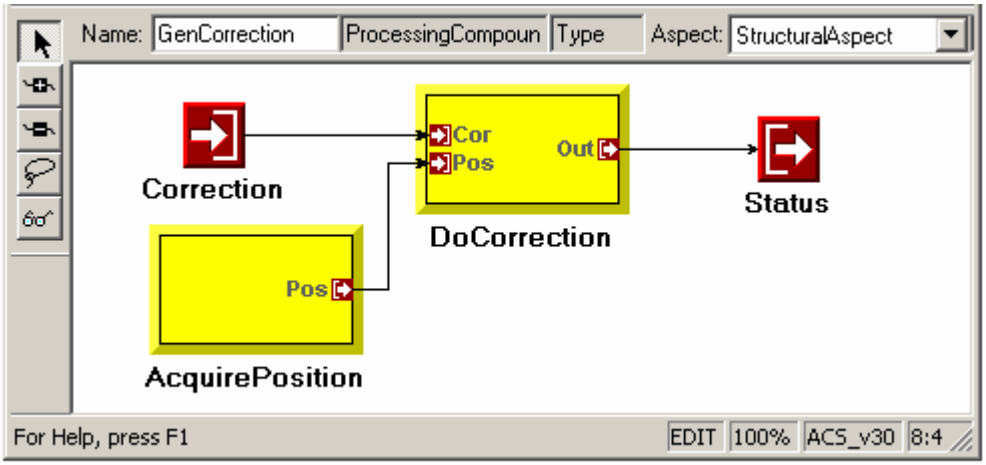

Figure 7. ProcessingCompound model GenCorrection, with submodels AcquirePosition and DoCorrection

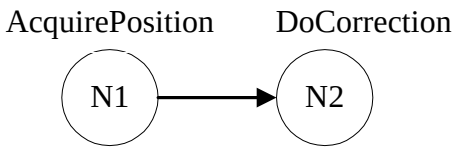

Figure 8. Graph representing the data dependencies of the GenCorrection model shown in Figure 7

Once the graph representing the data dependencies between the components has been formed, the static schedule may be derived through a topological enumeration

algorithm. A topological enumeration attempts to find an ordering of the nodes of a directed graph such that the next node selected in the ordering would have no input connections if all other previously selected nodes were removed from the graph, along with their incident edges. In the case of the graph in Figure 8, node N1, representing the AcquirePosition component, would be selected first, followed by node N2, representing the DoCorrrrection node. Node N2 could not be selected first, because it is the destination of a connection. In contrast, after node N1 is included in the enumeration, removing it from the graph along with the edge it sources leaves node N2 with no associated connection. Node N2 can then be selected. Figure 9 shows an example of a more complicated graph and one possible topological enumeration.

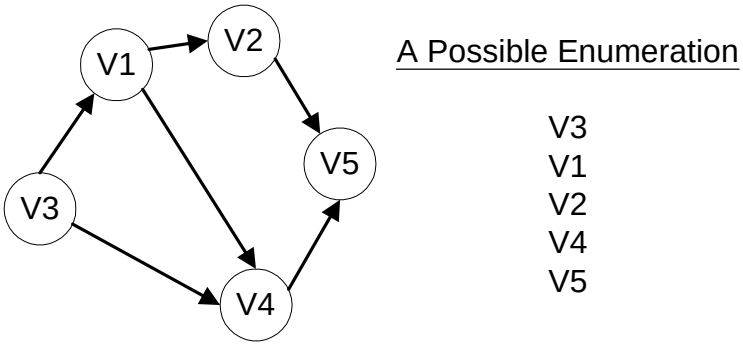

Figure 9. A more complex directed graph, with one possible topological enumeration

Because the topological enumeration enumerates the nodes of the graph in the data-dependent order, it represents a static schedule for the functions corresponding to the nodes of the graph. In the case of Figure 8, the node representing the AcquirePosition component was selected first, so the function representing the AcquirePosition component will be scheduled first, followed by the function representing DoCorrection.

MatSim writes calls to these two functions to a file in that order. This file becomes the function representing the GenCorrection compound.

Parameter Naming and Passing

As mentioned previously, the ports of a model represent the parameters of the model. The names of the input parameters of a function representing a compound correspond to the names of their assigned ports, likewise for the output parameters. The input and output parameters of a function are known as formal parameters. The actual parameters of a function are the parameters which are passed to a function when it is actually invoked. A connection leading from an input port of a compound represents a use of the formal parameter representing that port. If the destination of such a connection is an input port of a component contained in the compound, the formal parameter of the compound's function is used as the actual parameter in the call to the component's function.

Connections between models represent data exchanged between components. Because data is held in variables in Matlab, MatSim must generate temporary variables to hold this data. MatSim generates a temporary variable for every connection in a compound, except for those connections which begin at an input port, in which case the input parameter is used directly. Temporary variables are named according to the name of the port which sources the connection. This is only a naming convention; any temporary name could have been used. A globally unique integer is generated for each parameter and is appended to the end of the parameter name to guarantee uniqueness of the name. When a function generates an output which is to be used as an input to another function, the temporary variable is used as an actual output parameter for the function

call representing the source of the connection, and as an actual input parameter for the function call representing the destination of the connection. For example, in the case of Figure 7, there is a single connection connecting the AcquirePosition model to the DoCorrection model. This connection will cause the generation of a temporary variable, named after the output port, Pos, of the AcquirePosition model. The temporary parameter could therefore be named "Pos_1", assuming the global unique integer contained the value 1. Pos_1 will be used as an output parameter in the call to the function represented by the AcquirePosition component, and as an input parameter to the function represented by the DoCorrection component.

After all function calls have been generated for a file representing a particular compound, the formal output parameters of that compound must be updated properly. After the last function in the schedule of a compound has executed, the temporary variables, which represent the connections to the output ports of the parent compound, are assigned to the formal output parameters. For example, the code generator will insert a call to the function representing the DoCorrection model in Figure 7. This call requires one output parameter, so the code generator uses a temporary variable, named after the output port of the DoCorrection model. When the code generator discerns that no more functions remain to be called, it will update the output parameters of the function being generated by assigning the temporary variable generated in the call to the DoCorrection function to the formal output parameter Out.

Generating a Complete Function

```
function ResolveComponent( comp )

   If comp.ComponentType == ProcessingCompound
      Create file named after comp.name
      Write function prototype generated from the port interface of comp
      Write function instance counter
      For c:= next in the topological enumeration of the models contained in comp
          Find names of actual input parameters for input ports of c
          Generate temporaryies as actual output parameteres for output ports of c
          Write a call to the function corresponding to c
          ResolveComponent( c )
      Write updates of formal output parameters of comp
   end If
end function
```

Figure 10. Pseudocode description of the code generation algorithm used by MatSim

Figure 10 depicts a pseudocode description of the algorithm used in MatSim to generate a complete Matlab function from a compound model. For a model comp, MatSim first determines that the model is actually a compound model, because code is not generated for primitive components. It creates a file corresponding to the name of the compound model, appending the letter "M" to the front of the name, just as with primitive components. The name of the function generated for comp is the same as the name of the file, as required by the Matlab language. MatSim next discerns the proper function prototype for comp from the port interface of comp, being sure to account for the order of the formal parameters, as dictated by the port numberings. Next, a global variable which tracks the number of invocations of the function is written. The need for this variable will become clear in a later section. Following the instance counter, topological enumeration of the models contained in comp is performed, with the purpose

of discerning the order in which to invoke the functions corresponding to the contained models. As a component c contained in comp is encountered in the enumeration, it is taken as the next function to invoke in the schedule. To generate a function invocation, MatSim must determine the formal input parameter names to use in the function call. This is done by traversing each connection connecting to an input port of c to the source of the connection. Assuming the topological enumeration is performed correctly, the source of the connection will either correspond to an input port of comp, in which case a reference to the formal input parameter associated with the input port is used as the actual input parameter, or the connection source will be an output parameter of another model contained in comp which will have already been scheduled. The actual output parameters are generated as temporary variables, as described above. These temporary variables will be used either as actual input parameters in calls to yet-to-be-scheduled components contained in comp, or to update the formal output parameters of comp at the end of the function. After the actual input and output parameters have been gathered, a call to the function corresponding to c is generated and written to the file, and care is taken to ensure actual parameters appear in the proper order, according to the numbering of the ports of c. The ResolveComponent function is then invoked on c, so in the case where c is a compound, the function corresponding to it will be generated. After the loop through the topological enumeration completes, the formal output parameters of the function corresponding to comp are updated by writing an assignment involving the temporary variables corresponding to the connections to the output ports of comp. An assignment statement is written to copy the contents of the temporary variables to the formal output parameters.

```
function [Status] = MGenCorrection(Correction)

global MGENCORRECTION_INSTANCE_CNT;
[Pos_4] = MacqPos;
[Out_7] = MdoCorrection(Correction, Pos_4);
Status = Out_7;
MGENCORRECTION_INSTANCE_CNT=1;
```

Figure 11. Function generated by MatSim representing the GenCorrection compound shown in Figure 7

Figure 11 shows a function generated by MatSim representing the GenCorrection model shown in Figure 7. The function name and file name correspond to the name of the model, with the "M" added to the front: MGenCorrection. There is a single output parameter, Status, corresponding to the single output port of the GenCorrection model, and a single input parameter, Correction, corresponding to the input port of the model. The next line corresponds to the declaration of the global instance count variable. This is followed by the calls to the two functions representing the models contained in GenCorrection. The AcquirePosition model is selected first in the topological enumeration, so a call to the function realizing the AcquirePosition component is generated. The scriptname attribute of the AcquirePosition primitive model has been set to acqPos, so MatSim generates a call to a function named MacqPos. MatSim passes no input parameters to the MacqPos function because the AcquirePosition model contains no input ports. The data generated from the invocation of MacqPos is stored in a temporary variable named Pos_4, named after the output port of the AcquirePosition model. MatSim next selects the DoCorrection in the topological enumeration, and generates a call to the function represented by that model. The scriptname attribute of DoCorrection is set to doCorrection, as was shown in Figure 5, so MatSim generates a call to the function MdoCorrection. The actual parameters of MdoCorrection are passed according

to the port interface of the DoCorrection model. The first port of the DoCorrection model is connected to the Correction port of the GenCorrection model. This connection is represented as the use of the formal input parameter Correction as an actual parameter in the call to MdoCorrection. The second input port of DoCorrection is connected to the single output port of the AcquirePosition model, resulting in the use of the temporary variable Pos_4 generated in the call to MacqPos as the second actual input parameter in the call to MdoCorrection. The single output of the MdoCorrection function is stored in a temporary variable called Out_7. Because there are no more models to be processed, MatSim now generates code to update the formal output parameters of the function before terminating. The penultimate statement represents the connection from the output port of the DoCorrection model to the Status output port of GenCorrection. The final line of the function updates the instance counter, flagging that the function has been invoked at least once.

A Complete Functional Simulation

MatSim iterates over the hierarchy of models and generates a function for each compound in the hierarchy. When MatSim generates the code for the top level component, it not only generates the function calls as described, but it wraps the function in a loop which executes based on the condition of a global variable called TERMINATE_SIMULATION. A user may terminate the execution of a functional simulation by setting this variable to some non-empty value in a one of the primitives executing in the simulation. This allows the developer to control when the simulation terminates. After all functions have been generated, the user may execute the functional simulation in the Matlab environment simply by calling the function representing the top

level model in the application model set to be simulated. Executing this function will repeatedly invoke each of the other generated functions by tracing through the hierarchy. Each primitive function will eventually be invoked by the function generated from the compound containing the corresponding primitive model. By providing a function to represent the behavior of each primitive, the designer can invoke MatSim to generate a complete functional simulation of a modeled system.

Scheduling Feedback

The generation of a static schedule depends on the ability to represent the data dependencies between models as a directed graph, and the correct execution of the topological enumeration algorithm. When a model contains a feedback connection, the corresponding dependency graph will contain a directed cycle, causing the topological enumeration algorithm to fail. Figure 12 shows a top-level compound model named SimpleControl, similar to the model shown in Figure 3. The connection from the output of GenCorrection to the input of Comparison represents a feedback connection. While attempting to perform a topological enumeration on the graph representing the data dependencies between the models contained in SimpleControl, MatSim encounters the graph represented in Figure 13. MatSim can successfully determine that ReadSensorA and ReadSensorB can be properly scheduled. However, Figure 13 shows that the Comparison node is data-dependent on the GenCorrection node, and GenCorrection node is data-dependent on the Comparison node. MatSim cannot schedule one function before the other because it will violate the presumed data dependencies.

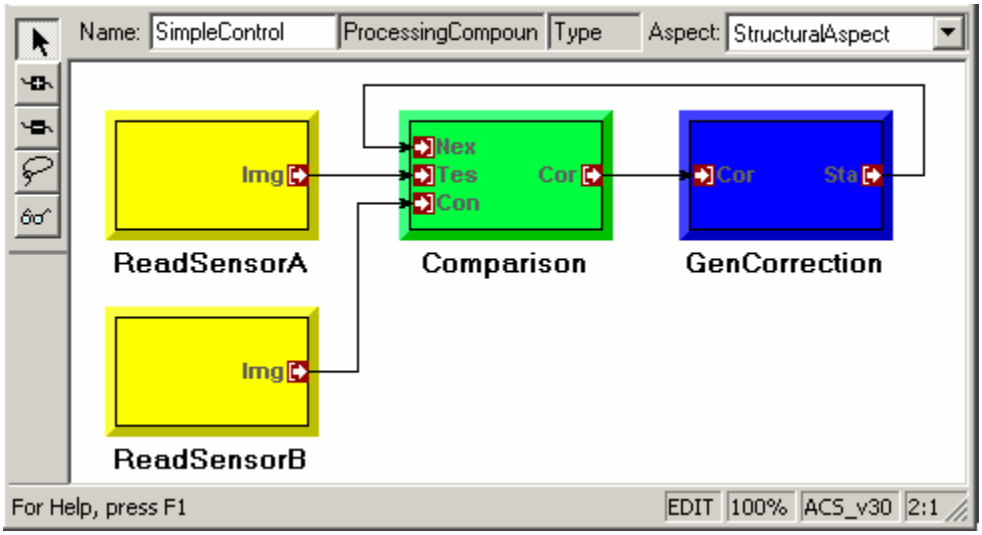

Figure 12. Compound SimpleControl, with a feedback connection from GenCorrection to Comparison

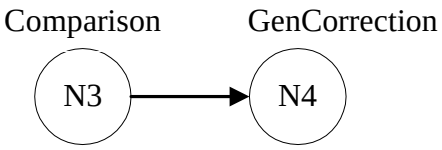

Figure 13. An unschedulable directed graph, caused by the feedback connection

However, this directed graph does not accurately represent the actual data dependency which feedback represents. In most applications, the component that receives data from the feedback connection does not receive data from the connection in the initial invocation of the function. That component simply generates an initial output and feeds the result forward in the network. Not until the component which generates the data to be fed back has had a chance to execute will the initial feedback data be generated. The dimension which is not shown in the directed graph is time. On the ith invocation of the Comparison component, Comparison depends on the data which was

generated in the i-1st invocation of the GenCorrection component. The GenCorrection component cannot be expected to produce an output prior to its execution, so the Comparison component must be constructed "knowing" an initial state for the feedback input.

In order to resolve the feedback connection in such a way that the proper results are computed in the functional simulation, the user must implement the Comparison component to "know" the initial state of the feedback connection. Further, the modeler must denote in the model which function "knows" to execute without receiving the initial data. This information is critical, because MatSim has no means of knowing if the Comparison component is the component which should ignore the cyclic input, or the GenCorrection component. The modeling language has been augmented with an atom to allow the modeler to make such a distinction. Figure 14 shows an updated SimpleControl model, with an initializer atom and connection in place. The initializer atom is simply a means for the modeler to specify to the MatSim interpreter where a directed cycle should be broken. By connecting an initializer atom to an input port of a model, the modeler is stating that that particular model has been designed to not depend on the input from the connected port on the first invocation. In other words, the component has been designed to provide its own initial data for the port connected to the initializer atom. On subsequent invocations, the function will be provided with data generated by the invocation of the function corresponding to the source of the feedback connection. With this updated information, MatSim can now determine that Comparison has been designed to ignore the initial input from the feedback connection, and can therefore be scheduled before the function representing GenCorrection.

Alternative, possibly superior, semantics that could have been applied to the initializer atom and the breaking of feedback loops is to allow the initializer atom to somehow provide an initial value for the port it is connected to. By allowing the initializer atom to actually initialize a port, a component does not need to be constructed with knowledge of how it is used in the topology of the interconnection. As described above, a component must be constructed to provide an initial value on the initial invocation, which is cumbersome from the perspective of a component library. It would be necessary to have a "feedback" and a "non-feedback" version of a component, even though both versions implement essentially the same function. However, the semantics in use were chosen because of the asynchronous dataflow semantics used in the ACS runtime environment. Because by nature components involved in asynchronous dataflow must be constructed to some degree to be aware of their own state, it is a small step to have them manage a small part of their own state. Assigning this second semantics to the initializer atom would require an update to the ACS runtime environment to properly implement the capability of initialization, while the assigned semantics are consistent with the runtime environment as it is currently implemented.

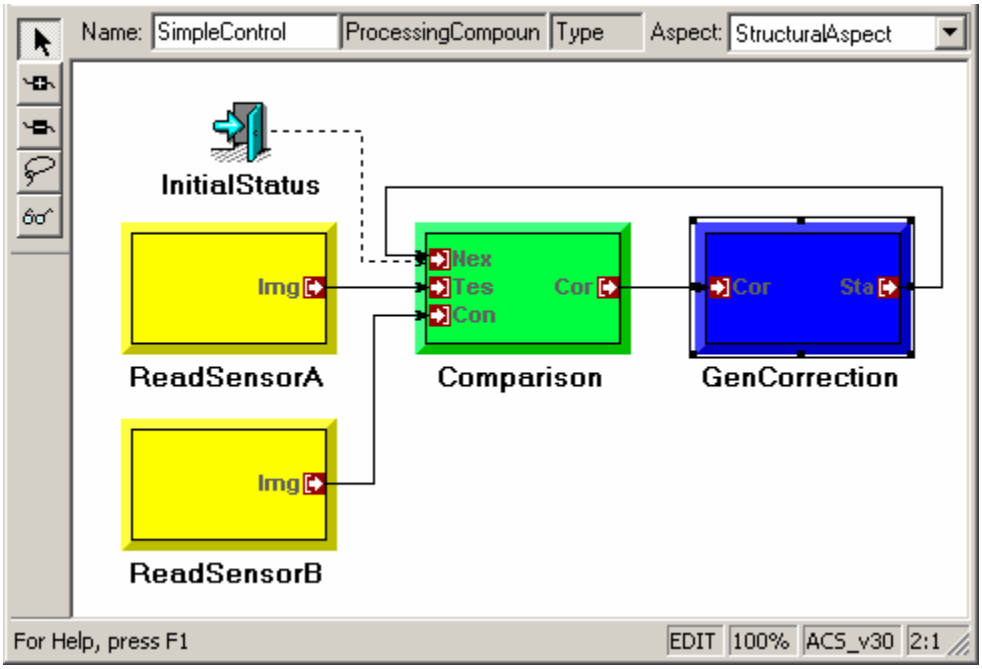

Figure 14. SimpleControl compound with Initializer atom and connection

MatSim must make a provision for passing a parameter which is involved in a feedback connection. When MatSim generates the call to the function representing Comparison, the input parameter representing the first input will not exist. The variable will be generated as an output parameter in the call to the function representing GenCorrection. This is not a problem, because Comparison has been written to ignore the input parameter on the first invocation. However, Matlab syntax dictates that some variable be passed as the actual parameter to the function on every invocation. MatSim therefore creates a variable to pass to the function. The variable must be initialized before Matlab will allow it to be used as an input parameter, so on the initial invocation of the function, the variable is set to the empty matrix. After the call to the Comparison function returns, the call to the GenCorrection function is executed, generating the feedback parameter which is to be passed to the Comparison function on the next

invocation. All temporary variables are created as local variables and are destroyed when a function returns to its caller. However, all feedback parameters must persist across invocations to properly implement the memory required by the feedback connection. MatSim therefore declares the temporary variable representing the feedback connection as global. Figure 15 shows the code MatSim generated from the SimpleControl model. The function is named the same as the model, and the function instance counter is declared. Because SimpleControl is a top-level model, the generated function calls are wrapped in a loop conditioned on the global variable TERMINATE_SIMULATION. The functions representing the ReadSensor components are called, followed by the statement declaring a variable called Status_3 as global. This variable is the parameter to be used to store the feedback connection results. The next statements will initialize the Status_3 variable to the empty matrix when the instance counter indicates that the function has not been executed previously. Next, the function representing the Comparison model is called, using Status_3 as an input parameter. This function will ignore the value of the Status_3 parameter during its first invocation. The next function call generates a value for the Status_3 variable to be used as input to the Mcomp function on the next iteration. Because this function represents a top-level model and is wrapped in a loop, technically it is not necessary to declare Status_3 as a global variable, because the variable will not leave scope before it is needed again. However, functions generated from models which are not at the top level will not be wrapped in a loop, thus requiring the feedback variable to be declared global.

```
function MSimpleControl()

global MSIMPLECONTROL_INSTANCE_CNT;
global TERMINATE_SIMULATION;
TERMINATE_SIMULATION = [];
while isempty(TERMINATE_SIMULATION),
   [Img_0] = Mread_sensA;
   [Img_1] = Mread_sensB;
   global Status_3;
   if isempty(MSIMPLECONTROL_INSTANCE_CNT)
      Status_3 = [];
   end
   [Correction_11] = Mcomp(Status_3, Img_0, Img_1);
   [Status_3] = MGenCorrection(Correction_11);
   MSIMPLECONTROL_INSTANCE_CNT = 1;
end
```

Figure 15. Code generated by MatSim representing the SimpleControl model displayed in Figure 14

Application: Bit-Width Simulation

When a designer explores different algorithms, it is often desired to perform an analysis of the effects of performing calculations with fixed-point arithmetic at various bit-widths. This type of trade-off analysis is most appropriate in a simulation setting, allowing a designer to determine the optimal bit-width required for a particular application, without needing to implement and test each solution.

MatSim provides a limited support to the designer to perform bit-width tradeoff analysis. The attributes of each port in the application models of a system contain information about the width of the data path to be used for that particular path. MatSim ensures that the widths specified in the source and destination ports of each connection are consistent. Matlab performs all calculations in double-precision floating-point format. However, between computations, MatSim can round parameters to the equivalent widths specified by the port attributes. The developer can choose to include rounding in a generated simulation by selecting the appropriate option in the user

interface of the MatSim interpreter. The rounding is performed by a function called roundfix, which takes a vector and a bit-width as inputs, and outputs the vector with each element rounded to the precision specified. Figure 16 shows the function generated by MatSim representing the GenCorrection model from Figure 7 with bit-width arithmetic simulation code included. The first statement after the function declaration is now a call to roundfix to round the formal input parameter Correction. After the call to MacqPos, the output Pos_4 is rounded as well. Each call to roundfix contains the parameter 16, representing the bit-widths specified in the model on each port. In this case, each port happened to be set to 16 bits.

```
function [Status] = MGenCorrection(Correction)

[ Correction ] = roundfix(Correction, 16);
global MGENCORRECTION_INSTANCE_CNT;
[Pos_4] = MacqPos;
[ Pos_4 ] = roundfix(Pos_4, 16);
[Out_7] = MdoCorrection(Correction, Pos_4);
[ Out_7 ] = roundfix(Out_7, 16);
Status = Out_7;
MGENCORRECTION_INSTANCE_CNT=1;
```

Figure 16. Code generated for GenCorrection model with fixed-point simulation code included

By including the fixed-point simulation in the functional simulation, the designer is allowed a somewhat more accurate view of what to expect during component execution on a fixed-point architecture. If the user provides components which accurately represent fixed-point arithmetic during functional simulation, a better fixed-point simulation will result.

A Comparison With Simulink

Simulink is a graphical system modeling package that allows a system to be represented as a set of block diagrams. In many respects, Simulink and the ACS modeling environment are very similar. As discussed in Chapter I, Simulink allows blocks to be hierarchically composed of simple primitive blocks, which are driven with Matlab functions. Simulink allows a designer to develop a simulation of a system quickly and efficiently. Simulink offers an extensive component library, covering many areas of engineering, including digital signal processing and control applications.

In a few respects, the ACS toolset extended with the MatSim model interpreter is superior to simply using Simulink to simulate the system, and then using the synthesis portions of the ACS toolset to generate the functional system. By integrating Matlab simulation capabilities into the ACS toolset, the developer may proceed with system design from a single system representation. Without MatSim, a designer is forced to recreate a design representation to support the simulation package of choice. While Simulink is a powerful simulation package, MatSim offers sufficient simulation capabilities to allow the designer to verify system models and perform high-level functional simulations. Further, Simulink does not support the Model-Integrated approach to system design discussed in this document, and therefore does not provide a designer with all the benefits of the ACS toolset. The merged capabilities of the ACS toolset and the MatSim interpreter allow a developer to apply simulation capabilities at design time, from a single design representation.

Functional Simulation Conclusions

MatSim provides the designer with the capability to generate a Matlab representation of system component models. This representation can then be executed, along with user-provided simulation components representing system primitives, to verify the models and experiment with algorithms.

There are a few drawbacks to this tool that have been discussed in the chapter. The simulations generated by MatSim do not precisely represent the semantics of process execution exhibited in the ACS runtime environment. MatSim assumes that the processes it translates follow the semantics of synchronous dataflow with all ports producing/consuming a single token on each invocation. Components that do not exhibit this behavior must be modified to maintain state to enable them to emulate this behavior or the simulation will not exhibit the correct overall behavior. If a graph exhibits a feedback loop, one of the components in the loop must be modified such that it provides an initial value for the feedback connection on the its initial invocation. The developer can specify which port of which component is to be initialized by the component by connecting an initializer atom to it.

While there are a few drawbacks, there are significant benefits brought to the ACS toolset by the MatSim interpreter. MatSim allows simulations to be generated directly from the system models, providing a single, integrated system representation from which the designer can start and complete the system design. Simulation components can be constructed as "standard" Matlab functions. A developer is not required to interact with a simulated runtime environment or an API when constructing component simulations. Potentially, any Matlab function can interface to a simulation

generated by MatSim. While the semantics of asynchronous dataflow are not exactly replicated in the simulations generated by MatSim, the claim has been made in this chapter that any differences between the semantics used by MatSim and those in the actual runtime environment are negligible at the level of simulation detail targeted by the MatSim tool. The strongest result is that MatSim allows a developer to apply model-integrated design techniques at the very earliest stages of a system design, during algorithm development and concept design, allowing the verification against design requirements early in the design.

CHAPTER IV

VIRTUAL PROTOTYPING

Chapter III discussed a tool to allow simulation to be utilized during model construction as a means of high-level design verification. In the model-integrated approach, after the system has been modeled, the next step in the system design is to design and implement the primitive components. This chapter discusses an extension to the ACS toolset integrating simulation into the design and testing of components. This extension is referred to as the support of virtual prototyping. By integrating simulation into the design flow at this stage, the developer is provided with a means to detect and correct faults in component designs earlier in the process, thereby saving time and effort later in design stages. In the context of the ACS toolset, a virtual prototype is a system prototype which integrates simulation components at runtime. With the virtual prototyping extensions, a developer may generate a completely simulation-based prototype, in which all components are simulation components. A developer can also generate a system where some of the components execute on their native implementation platforms, while others execute in simulation. As a consequence of the virtual prototyping extensions, components can be simulated in the context of an actual system implementation, and component implementations can be easily tested as part of a simulation.

Figure 17 shows the extended runtime architecture of the ACS toolset. The runtime system is configured and managed through software executing on a host PC. The virtual prototyping extensions to the toolset include the Matlab environment

executing on the Host PC, which can communicate with the management software via an interface called the Matlab Engine [12]. Matlab exports the Engine interface to allow standalone software packages access to the computational power of the Matlab environment. All operations which can be performed in the Matlab environment through its command line interface can be invoked by an external program via the Engine interface. Further, data may be exchanged between the external program and the Matlab environment. A program can invoke Matlab functions through the Engine interface as easily as it invokes native subroutines. By utilizing the Matlab Engine interface, the ACS runtime environment has been extended to allow simulation components to execute in the Matlab Environment and to exchange data with components executing in the heterogeneous processing network.

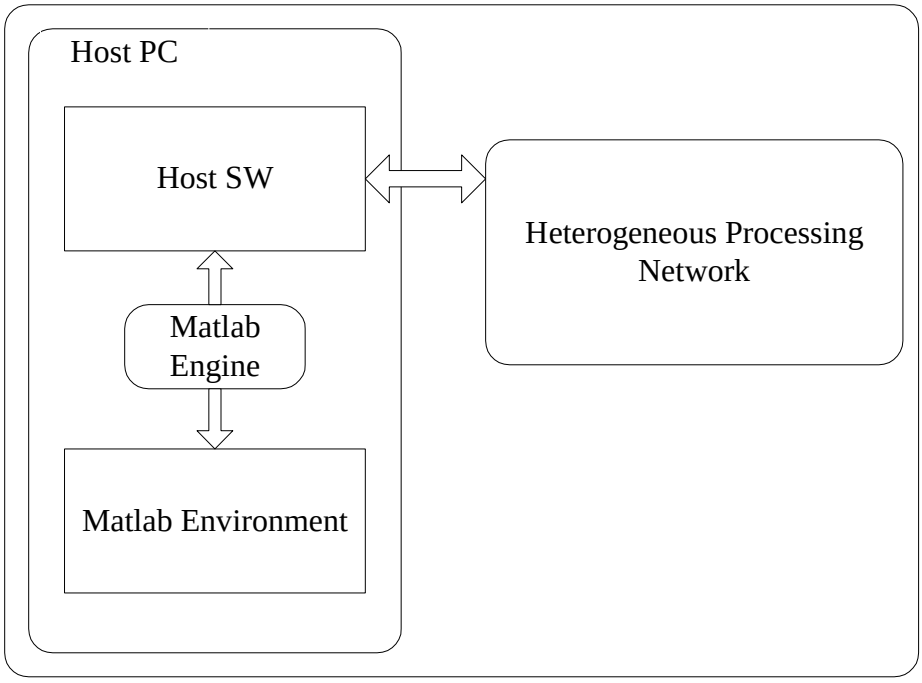

Figure 17. Architecture of the Virtual Prototyping Extensions to the ACS toolset.

The ACS toolset has been extended to support this runtime architecture. These extensions involved updates to the modeling environment and model interpreter, as well as extending the runtime environment to support the execution of simulation components at runtime. The following sections discuss the extensions which were made.

Extending the Modeling Environment and Synthesis Tools

Modeling Environment Extensions

The goal of the virtual prototyping extensions is to allow simulation components to execute as part of a system at runtime. To represent this concept graphically in the modeling tools, some minor modifications to the ACS modeling language were made. By allowing simulation components to be integrated into a system, a simulation component is, in a sense, equivalent to a system primitive component whose target implementation is targeted to a processing element in the heterogeneous network. In the same sense, the platform on which simulation components execute, namely the Matlab execution environment, is equivalent to a network processing element. Accordingly, the Matlab environment is modeled as a processing resource in the heterogeneous network. As depicted in Figure 17, the Matlab environment actually executes on the host PC of the system, and communicates with the host management software via the Engine interface. This architecture is modeled by allowing a model of a host PC to connect to a Matlab resource model. The modeling tools allow a PC resource to connect to a Matlab resource model, representing the fact that the only means to communicate with the Matlab resource is via the host management software and the Matlab Engine. The modeling tools do not allow any other types of resources to connect directly to the Matlab resource

model. Figure 18 depicts a set of resource models, showing a model of the Matlab environment as a processing resource model, connected to the model of the host PC. The host PC is in turn connected to the remaining resources in the network, in this case a model of a TMS320C40 DSP, a TMS320C67 DSP, an Altera FPGA, and an SDRAM module.

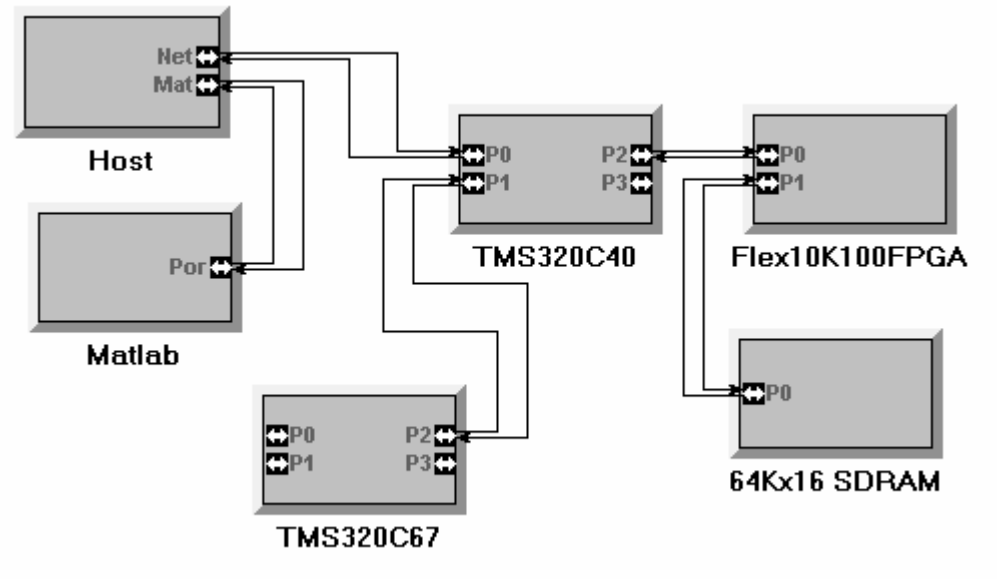

Figure 18. Resource Model showing the Matlab environment interfaced to the host

The modeling environment represents the communication protocol established between the Matlab environment and the Host management software via the Engine interface as a protocol named Matlab Protocol, as depicted in Figure 19. The Protocol attribute of the port of the Matlab resource model, along with the port of the Host resource model connected to the Matlab resource model, must be set to “Matlab Protocol” to correctly represent, and therefore synthesize, a virtual prototype system.

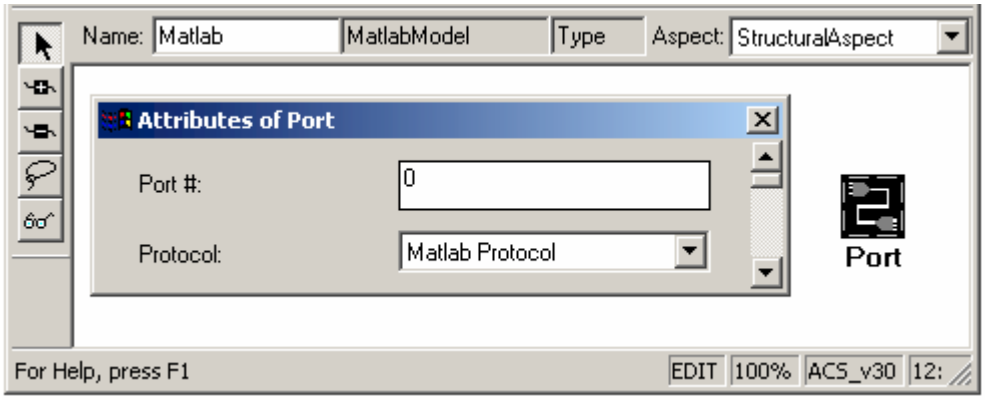

Figure 19. Port Attributes showing Matlab Protocol as the selected communication protocol

Simulation components are explicitly modeled as part of the component dataflow network. In contrast to the component simulations described in Chapter III, where all system components were assumed to be simulation components regardless of their associated resource mapping, an extension to the modeling paradigm allows a modeler to categorize a primitive component model as a Matlab Simulation component. Models of primitive simulation components can be mapped to the Matlab environment resource model, just as models of primitive DSP components can be mapped to a model of a TMS320C40 DSP. Because functions implemented cannot execute natively on any resource except the Matlab environment, the only resource model to which models of simulation components can be mapped is the Matlab resource model. By constructing models of simulation components, a developer may represent a virtual prototype system containing both simulation components and other executable components.

Extending Model Interpretation

In keeping with the model-integrated approach to system design, a goal of the virtual prototyping extensions of the toolset is to support the synthesis of a functional

virtual prototype system directly from the models. The ACS model interpreter has been modified to meet this goal. Because the Matlab environment is modeled as a kind of processing resource, the interpreter can simply treat it as a node in the heterogeneous network when generating middleware initializations and configurations. Some minor additions were required to generate initializations for the middleware layer implementing the runtime environment for the Matlab environment, however most of the configuration code was simply inherited from the existing interpreter code base.

Extending the ACS Runtime Environment

Chapter III described the synthesis of high-level simulations from dataflow models of components. One drawback to that simulation was the fact that the MatSim interpreter imposed a limitation on the types of system models it can simulate, due to dataflow scheduling issues. The reasoning behind this tradeoff was that it was not desired to require a developer to interact with a middleware layer which would perform dynamic scheduling and buffer management. The rationale behind this choice was that during the modeling phase of the design, a modeler is most concerned with representing and properly composing a system. Components are seen as black boxes that perform computations. At model building time, a developer is not concerned with how a component performs those computations, or with any implementation details of the component. The goal of the virtual prototyping extensions to the toolset is to support the integration of simulation into component design and implementation. Obviously, when components are designed and implemented, they are no longer treated as simple black boxes that perform computations. In this phase, it is much more important to understand the behavior of the component, in the context of the full system. As such, the virtual

prototyping extensions include a middleware layer to implement the runtime environment in the Matlab environment. As discussed in Chapter II, the runtime environment facilitates component scheduling. It abstracts the details of inter-component and inter-node communication from the components by servicing the communication needs of the component through an API. The middleware layer implemented for Matlab facilitates the communication between components executing in the Matlab environment and those executing in the processing network. The implementation of the Matlab middleware layer closely follows that of the runtime kernel for a stored-memory processor in the processing network. The following sections review relevant details of the ACS runtime kernel, followed by a description of how the runtime kernel features were implemented in the Matlab middleware layer.

The ACS Runtime Middleware

As discussed in Chapter II, each processor in the network runs a small dataflow kernel [11], which supports deterministic dynamic memory management, stream-based inter-process and inter-node communication, and process scheduling and management. Figure 20 depicts the layered architecture of the runtime kernel. The API layer allows components to access the services provided by the kernel, such as memory and stream management. Through the API layer, a component may determine the state of its input and output streams, thereby determining its schedulability. The API layer interacts with the stream, memory, and process management facilities of the kernel. As previously discussed, process management consists of a simple round-robin non-preemptive scheduler, successively invoking each component mapped to the node. The kernel memory management layer provides a memory pool from which processes may

dynamically allocate buffers. The stream management layer implements the concepts of dataflow streams connecting components, facilitating inter-component communication. Inter-node communication is facilitated through a set of functions called interface functions, which drive the communication hardware. As all communication links in the heterogeneous network are point-to-point links between communication ports, each communication port of a processor is assigned its own set of interface functions. The interface functions implement a communication protocol, which determines the format and organization of the data during transmission. Obviously, the two communication ports connected by a link must support compatible communication protocols, or data will not be transferred coherently. Interface functions are designed as a pair of functions, one responsible for the transmission of data, while the other for receipt. The transmission interface function for a port is invoked when a stream queue corresponding to that port contains a message to transmit. The receive interface function for each port is always active, awaiting the arrival of data. When a buffer is received, the receive interface function notifies the stream management layer, which dispatches the received buffer to the proper stream queue, determined by the message routing information passed with the buffer. When the transmission of a buffer is completed, the stream management layer is notified, whereon the buffer is removed from its stream's queue.

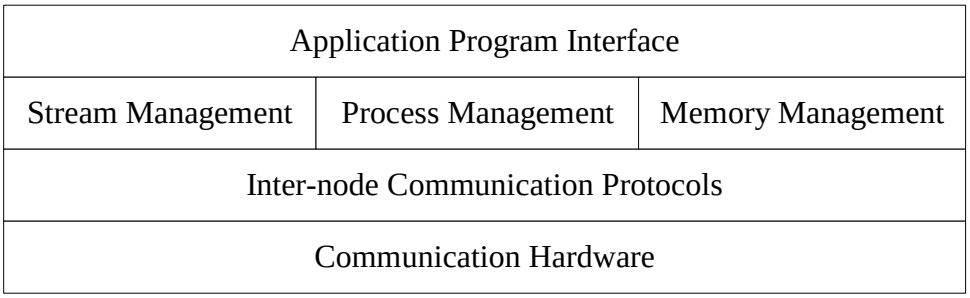

Figure 20. Layered architecture of the ACS runtime kernel

Matlab Runtime Middleware

A goal of the virtual prototyping extensions is to accurately represent the execution semantics of components at runtime. To facilitate the accurate representation of dynamic scheduling and message passing, a runtime middleware layer was implemented for the Matlab environment. The middleware implements a layered architecture, shown in Figure 21, similar to that of a processor runtime environment. However, there are some minor differences. There is no communication hardware between the host management software and the Matlab environment. It has been replaced by a Host Interface layer, which interacts with the interface functions in the host management software via the Matlab Engine to facilitate data exchange. Dynamic memory management is not required in the Matlab environment, because in the Matlab environment, all memory allocation is dynamic and is handled by the environment. No gains can be made by imposing another management layer on top of the environment. Process management is handled in much the same way as the processor runtime environment, as is stream management. The API layer allows simulation components to access the stream management facilities of the middleware. The layered architecture for the Matlab middleware is depicted in Figure 21. The middleware implementation

allows components to be dynamically scheduled in the same manner as components executing on a network processor are. Just as with a network processor, the details of inter-component and inter-node communication are abstracted away from the component. All the component sees is the API layer provided by the middleware.

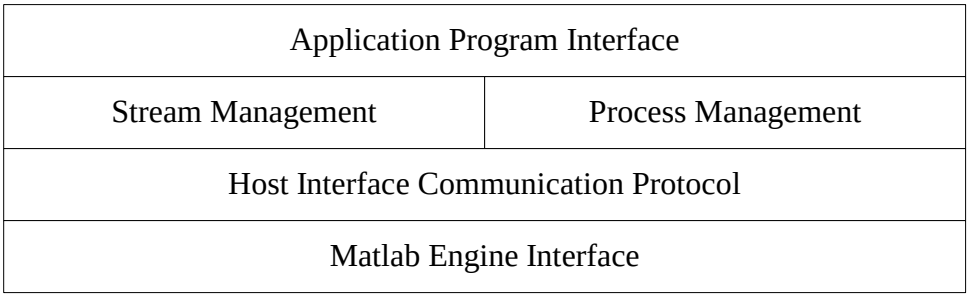

Figure 21. Layered architecture of the Matlab middleware

The Matlab Communication Protocol

The architecture described facilitates the execution of simulation components at runtime. Since the goal of the virtual prototyping extensions is to facilitate the exchange of data between simulation components and network components at runtime, a communication protocol has been implemented to take advantage of the Matlab Engine interface to facilitate data exchange. Before describing the details of the host and Matlab interface functions, a few points should be noted. The Matlab Engine interface provides a unidirectional access path, meaning that the host management software can invoke methods of the Matlab middleware, but the reverse is not possible through the Engine. While the Engine provides access to the Matlab environment for a standalone program, the Engine does not support multithreading, in that, when the host invokes a method in the Matlab environment, the host is blocked until that method returns. The semantics is

the same as when the host invokes a native function. Due to these limitations, the Matlab middleware has been designed to operate as a slave to the host management software. When the host transmit function is invoked to send the Matlab environment a buffer, the transmit function must not only place that buffer in the Matlab workspace, but must invoke the Matlab middleware receive interface function to allow the middleware to properly store the received buffer. Likewise, the host receive function must invoke the Matlab send interface function to allow the middleware to transmit data to the host. Further, in order to allow simulation components to execute, the host has been designed to periodically invoke the Matlab middleware process scheduling facility. This triggering of the process management occurs at the end of every invocation of a host interface function, thereby avoiding the starvation of the Matlab components.

The interface functions are responsible for exchanging data with the Matlab environment. Data is exchanged between processes in the form of messages. A message consists of a header and a body, as depicted in Figure 22. The header contains the routing information for the message, containing the handles of the node and stream where the message initiated, as well as the handles of the intended destination node and stream. A message in the Matlab execution environment is represented as two vectors. Each field of a message header has a corresponding index into a header vector. A message body in the host kernel is represented as an array of numbers. This is consistent with the Matlab representation: a vector of numbers. When the host transmit interface function prepares to send a message to the Matlab kernel layer, it allocates two vectors in the Matlab workspace via the Engine, one vector for the header and another for the body. The data from the host header is then copied, field-by-field, into the header vector. Similarly, the

message body is then copied into the body vector. However, an issue arises when copying data from a kernel message into a Matlab vector. The basic data type in the Matlab computational environment is double precision floating-point, requiring all fields to be cast to doubles as they are copied into a Matlab vector. When a message body is copied to a body vector, the interface function must make an assumption about the current data format of the host message body. Regardless of the explicit type declared in the message body data structure, a component can store in a message data in an arbitrary format, so long as the source component is consistent with the destination component. However, the interface functions between the host and Matlab must make an assumption about the format of the data contained in a message body, because it must convert the data into double precision format. By convention, the interface functions assume that all message bodies are currently stored in single-precision floating-point format, and the responsibility for ensuring that this is the case is placed on the system developer. Because there is no characterization of the type of information being passed in a buffer, the interface functions are left with no other choice but to assume a format and perform the conversion.

Figure 22. Architecture of a message

Once the host transmission interface function copies the message header and body into Matlab vector variables, the Matlab kernel layer receive function is invoked, passing the header and body vectors as parameters. The Matlab receive interface function simply

transfers the received message into the stream management layer, which dispatches the message into the proper stream queue. The Matlab receive function then returns control to the host. The host then frees the vectors which were allocated in the Matlab workspace, and invokes the Matlab middleware process management routine. On return from the process management, the host transmit interface function returns control to the host kernel.

The host receive function is very similar to the send function, only it performs its actions in the reverse order. The host receive function first invokes the Matlab kernel send interface function. If there is a message awaiting transmission in the Matlab middleware, the send function retrieves it from the stream management layer and returns it to the host. The host receive function then verifies that an actual message has been sent, then allocates a buffer from the host memory management system and copies, field-by-field, the header vector and body vector into the allocated buffer. Just as before with the host transmit function, the host must assume the format the message body is supposed to be in is single precision floating point. When the message is successfully copied, it is passed to the stream management layer of the host kernel. After storing the copied message, the receive function invokes the process management function of the kernel layer, and then returns.

These interface functions allow the Matlab execution environment to exchange data at runtime with processes running on the network. The model interpreter facilitates data forwarding across the nodes in the network to handle the situation where a process executing on a DSP in the network generates a message for a process executing in the Matlab environment. The generated message is forwarded to the host management

software on the Host PC, where it is subsequently transmitted to the Matlab environment. A similar process is performed for the reverse direction of communication. This communication framework allows data to be exchanged between components executing in the Matlab environment and any other component in the network.

Virtual Prototyping

With the extensions to the runtime environment and modeling tools, a developer can now construct a virtual prototype of a system. A system can be modeled as a set of simulation components, which can be implemented using the Matlab language and kernel layer API. A functional system can be synthesized from the models with the model interpreter, and can be loaded onto the resource network. During initial design stages, this resource network could consist of a host PC with the Matlab environment. After compiling and loading the code, the developer may test the system to verify its behavior. After the behavior has been verified, the resulting system represents a virtual system prototype, exhibiting the functionality of the target system, but implemented using a simulation language. The virtual prototype will obviously not meet the performance requirements of the target system, but will demonstrate the core behaviors of the target.

After the virtual prototype has been constructed, it can be used during the component implementation design phase. The modeling tools automate the selection of implementation alternatives from the models. When a system is modeled, alternative implementations for each component can be explicitly included in the models. As one alternative implementation for a component, the user provides a Matlab-based simulation implementation. A second alternative implementation is the actual target implementation. When constructing the virtual prototype, the target implementations for

system components need not be constructed, nor even modeled. However, the user should make use of template models to allow alternative implementations to be modeled later. With a system modeled in this fashion, a designer can select a full simulation-based implementation for the system and generate a virtual prototype with the synthesis tools. The virtual prototype can then be executed and verified against the functional requirements of the system. After verifying the correctness of the virtual prototype, the developer can use the prototype as a framework for testing component implementations. When a component is implemented, the tools can be used to select the simulation implementations of all system components except for that particular component, whose target implementation is included in the final design. After this design is synthesized and loaded, the user can verify that the component's target implementation exhibits the same behavior as the simulation implementation. The virtual prototype provides an ideal testing framework because each simulation component has, at this point, already been verified and the simulation components can be used to manipulate the inputs and display the outputs of the component under test. Each component is tested in the context of the system, and the designer is saved the effort of building a testbench framework for each component.

The virtual prototype also provides an excellent framework to perform system integration. As components are implemented and tested, they may be integrated into the prototype system one at a time, replacing their simulation-based counterparts. As more components are included in this system, integration issues may be uncovered. Previously, integration issues could not be thoroughly examined until most or all of the components had been implemented and could be included in a synthesized system.

Because the virtual prototype system allows components to be integrated seamlessly, integration issues can be examined much earlier in the design phase. Systems can be synthesized where some of the components are simulation-based, while others are implemented as their target implementations. Through testing these systems, the designer can examine how components interact in a more controlled environment.

Matlab components can be used as a debugging tool. Because network components can exchange data with Matlab components at runtime, the designer can insert a Matlab component in the data path between two components to visualize the contents of the messages being exchanged. Matlab components can also be used to modify the inputs to a network component, allowing a greater versatility in testing.

The user is provided with a powerful tool to perform verification, debugging, component testing, and system integration through virtual prototyping. By providing an interface between the Matlab computational environment and the processing network, a developer is allowed to utilize the power of Matlab at runtime, intermixed with implemented components.

CHAPTER V

AN IMPROVED DESIGN FLOW

The extended ACS toolset provides an improved platform for designing embedded systems. The addition of simulation capabilities to the toolset leads to an improved design flow, which leads to better, more efficient system development.

Analysis of ACS Design Flow

The ACS design flow allows a developer to build models and synthesize systems from those models, as was discussed in Chapter II. Figure 23 is similar to the design flow presented in Chapter II, depicting the different phases of design when using the ACS toolset to develop a system. First, algorithms and ideas are developed and explored, possibly constructing a simulation prototype, using other tools. This development builds a high-level representation of the system under development and allows the developer to explore different concepts to be included in the design. After completing this exploration step, a developer then uses the modeling environment provided in the ACS toolset to build and refine system models, breaking complex systems into simple, modular components. These models form a second representation of the system, graphically capturing system requirements and specifications. After the models are developed, the components are individually implemented and tested. When all components have been implemented, the developer may synthesize a complete system from the models using the model interpreters. This synthesized system provides all “glue code” to connect system

components and handle inter-component communication. This system can be loaded onto the resource network and tested for integration issues.

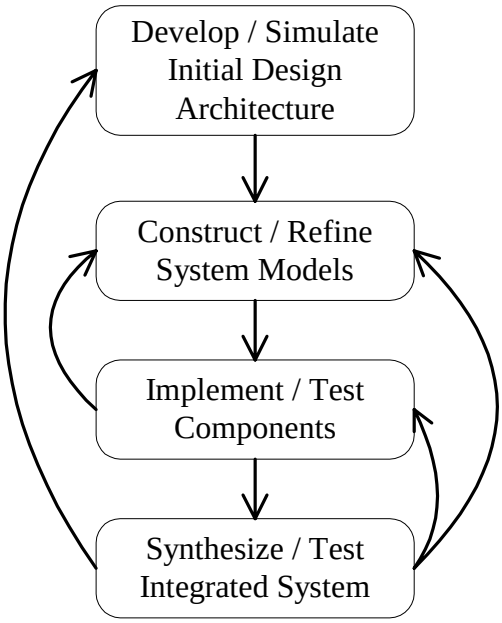

Figure 23. ACS system design flow

While a developer can use the ACS toolset to design complex systems, some issues with the design flow arise through design iteration. First, the construction of a high-level functional simulation is not supported directly by the tools. The modeler has no direct means for checking system models against functional requirements prior to system implementation. If a design flaw is introduced when modeling the system and is not caught until system integration, the effect of the flaw could possibly propagate across several component implementations. As a consequence, the correction of the flaw requires significant time and effort, which could be completely avoided by catching such flaws at model building time.

A second issue with the ACS design flow can be seen during system integration. Full integration cannot proceed until all components have been implemented and individually tested. Often, integration can uncover inconsistencies between components, requiring component adjustment or redesign. Iteration during this phase in the design is costly, but can be minimized by enforcing consistent component interfacing and synthesis of glue code for integration. However, design errors will inevitably occur, so a better approach is needed to catch such errors as individual components are implemented.

A final issue with the ACS toolset is that no support is provided for high-level design verification. The modeling language and environment enforces a designer to be consistent in design entry, and to have syntactically correct models. However, the language cannot enforce correct semantics. Simulation at model building time can add aid the developer to verify design semantics before proceeding to implementation. Simulation at runtime can facilitate system debugging.

Analysis of the Extended ACS Design Flow

By adding the simulation capabilities to the ACS toolset described in this thesis, an improved design flow has emerged. Figure 24 depicts this improved design flow. The first step is to construct and verify system models. This step proceeds from a high level of abstraction, beginning with a model of the initial design architecture. Simulation of this initial system architecture model is facilitated through the MatSim interpreter. The designer verifies the high-level architecture against system requirements. The designer proceeds to recursively decompose the high-level architecture into simpler pieces. At each step in the decomposition, the modeler may generate a functional simulation to verify the modeled system against the requirements. By verifying the system models as

they are constructed, the modeler can catch design flaws early in the process, prior to proceeding to the component design and implementation phases.

The next step in the design process is the design and simulation of system components. The virtual prototyping extensions to the ACS toolset are utilized at this stage to create a simulation of the system which accurately represents the execution semantics of the actual runtime environment. To construct a virtual prototype, system components are modeled as implementation alternatives, with a simulation implementation as an available alternative. The developer constructs a simulation-based component for each system component, and the tools are used to synthesize a simulation-based system by selecting the simulation implementation of each system component. This simulation-based system can be executed and tested, allowing the developer to verify the behavior of each system component. The developer spends time at this stage to ensure the virtual prototype correctly exhibits the specified behavior (except for performance) of the target application. If any inconsistencies are uncovered through the development of the prototype, the models are adjusted and prototype development continues. After the virtual prototype has been verified, the developer is left with an accurate simulation of each component in the system. These simulation components act as a design specification for the later state of component implementation. The next design step is to implement, test, and integrate system components. At this stage, the developer has a correct virtual prototype of the full system. As described in Chapter IV, a consequence of the automated selection between design alternatives is the ability to synthesize a system consisting of a single component implemented for its target platform in the processing network, with the remaining components implemented in simulation.

This system can be executed and verified. The advantage of such a system is that the virtual prototype acts as a testbench for the single component implementation. The implemented component is effectively integrated into the pre-verified simulation, allowing the component to be tested in the context of the full system using application data. This process is repeated for each component, allowing the testing of components to proceed in a semi-automated fashion. Further, as components are implemented, partial system integration can be performed, swapping simulation components for implemented components. In this fashion, as components are designed and implemented, system integration issues may be examined at a much earlier stage than previously possible. The virtual prototype provides not only a framework for verifying design semantics, but a framework for testing component correctness, and a framework for partial system integration. It also provides a unique environment for debugging a system. Simulation components can be used to quickly visualize data between components, as well as to inject data into components at run time, thus allowing a developer the ability to "zoom in" on problems in a system.

The last step in the design is to test a final system. This is performed after all components have been implemented and integrated. At this stage, a design must be tested to verify that it meets performance requirements, and any other issues which could not be addressed through the testing with the simulation components "in the loop."

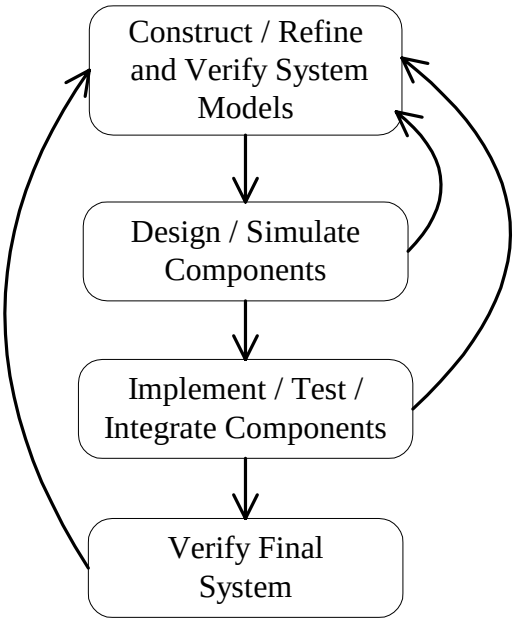

Figure 24. Improved design flow for extended ACS toolset

The improved design flow allows a designer to more effectively design systems. By integrating the complete design flow into one tool, the designer can develop systems from a single design representation. The simulation capabilities streamline the design process at each stage of development, allowing design flaws to be caught early, removing the tendency towards design iteration discussed in Chapter II. Iteration in system design is in general inevitable, because mistakes will be made. However, in this improved design flow, catching flaws and errors earlier in the design process minimizes iterations.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

A model-based approach to embedded system development aids the design of complex embedded systems. The work presented here enables the use of simulation techniques in conjunction with model-based design methodologies when constructing complex embedded systems.

MatSim: A Functional Simulation Generator

A functional simulation of a system is an executable system representation, which exhibits, to one degree or another, the behavior of the final product. A design engineer can construct a functional simulation as part of an initial exploration of a design concept. When thinking on this level, a designer is not concerned with, and therefore should not be burdened with concern for, hardware architectures or platforms. The designer can use the functional simulation capabilities discussed here to verify a set of system models. MatSim allows a designer to write Matlab code representing the functionality of a component using the Matlab language. The designer does not need to be aware of any API or kernel, and can do quick computation and analysis using the computation and visualization capabilities which the Matlab environment provides. All the glue code connecting components is synthesized by the tool, and all components are scheduled properly by the tool, so the designer need only be concerned with expressing the behavior of the system components.

By allowing the designer to generate an executable functional simulation from the system models, the tools not only provide a solid basis for experimentation, but models used in constructing the functional simulation can be reused in successive design phases. When system models are updated, a new functional simulation can be regenerated by adjusting the simulation components and executing the code generator on the updated models. High-level functional simulation can be used to verify a high-level design prior to implementation.

The MatSim interpreter was utilized in the design of two real-world applications, an automatic target recognition system for missile guidance, and a probabilistic neural network system used for image classification. The tool was found to be very helpful in quickly producing Matlab simulations of the systems, allowing the visualization of high-level behavior and functionality. The plotting and data visualization routines provided by Matlab proved highly valuable at this stage. However, the restriction placed on the modeling semantics by the MatSim tool was found to be cumbersome when constructing component simulations. The tool proved highly useful in simulating the effects of fixed-point arithmetic on algorithm correctness. The target platform for the image classification application is a set of FPGA nodes, with all operations performed in fixed-point arithmetic. By simulating the effects of datapath width variations, a reduction in the number of gates required to implement the system was achieved. Overall, MatSim proved to be a useful tool for generating simulations during system modeling.

A Virtual Prototype

The functional simulation generated by the MatSim model interpreter provides a complete, executable virtual system prototype. However, this prototype is essentially

used for high-level simulation at model-building time. A stronger virtual prototyping mechanism is provided by allowing Matlab-based components to be run "in the loop" at runtime with components executing on their native platforms.

This virtual prototyping mechanism provides the designer with a powerful system integration and debugging tool. When the models have been constructed and the designer begins implementing the system components, the virtual prototype using Matlab-based components and a middleware layer to facilitate communications with the network runtime environment provides a framework for system integration, as well as a testbench for each component. When a component implementation is introduced into the system, it can be tested in the context of the final system framework by replacing the Matlab version of the component with the final implementation version. This new system can then be tested, just as the prototype itself was tested. Inputs to the component can be easily generated, and outputs are readily visualized through the remaining Matlab components. By having the virtual prototype act as the testbench for each component in the system, not only does the designer alleviate the task of reproducing testbenches for each component, but the components are arguably tested in a better context, the final application.

After individual components have been verified, they can be used in the prototype along with other implemented components, interfacing to the virtual prototype. In these stages, combinations of components can be tested together, to check for system integration issues. Using the new virtual prototype as an integration framework, system integration issues can be examined as components are implemented, rather than waiting

until all components have been implemented to test the integrated system. This allows errors to be caught and addressed earlier in the design process.

Virtual prototyping was applied to in the development of the neural network image classification system, as well as the automatic target recognition system. The ability to execute simulation components concurrently with a system implementation proved highly useful in design implementation. It was seen, as expected, that operations performed by the simulation components were much slower than those performed on the embedded processors. However, virtual prototyping enabled the use of data visualization routines at runtime. Further, the ability to integrate system components into the simulation proved useful, providing flexibility in component implementation. The data conversion restriction imposed by the virtual prototyping interface proved cumbersome, as it was always necessary to perform conversions, whenever data was to be sent to the Matlab environment. The ability to visualize the state of a system at runtime proved very useful as well. Overall, the virtual prototyping mechanisms allowed component development to proceed more smoothly, and provided a useful framework for system debugging.

Future Work

While the extensions to the design tool allow certain simulation techniques to be applied to the design process of embedded systems, there are some areas where more work could be done.

Currently, when a virtual prototype component is interfaced to a network component and an exchange of data takes place, the interface assumes that all data to be sent to the Matlab component is in single precision floating point format, and that the

data being sent to the network component should be converted to single precision floating point format. This assumption was made because Matlab performs all calculations in double precision floating-point format. There is not currently a convenient way to discern what format a buffer of data should be in. The current assumption is that it is the designer's responsibility to handle any data formatting issues, but a better solution to this problem could be explored, involving a finer-grained modeling of the type of data being exchanged between components.

MatSim provides limited support for exploring precision effects on algorithm mathematics. However, this support is provided as a call to a vectorized truncation function, which rounds results to the bit-widths specified by the port attributes in the models. This solution is not adequate for two reasons. The first is related to the problem with implicit conversions mentioned above: if the user chooses to pass data between components in the form of a data structure, the vectorized truncation function will not work. The toolset needs a better idea of the type of data to be used in parameter passing. The second reason is it is very difficult to perform a tradeoff analysis, because when attempting to adjust the widths used in an algorithm, it is necessary to adjust the attributes on each port of each component involved in the change. A better, more modular approach to specifying bit-widths in the models is needed.

In a more general sense, further simulation techniques could be applied during system design. Currently, all simulations execute through the Matlab environment. A new type of simulation could be introduced into the system which integrates a VHDL simulation with a Matlab simulation. This would allow a designer to execute simulations of hardware components based on their native language, along with the simulations of

other components. This would require the interfacing of a VHDL simulator to either the MatSim interpreter or to the runtime environment. In a similar context, it could be possible to interface an instruction set simulator for a processor to the tools as well, allowing a designer to simulate most aspects of a system based on the same set of models.

REFERENCES

[1] M. Moore, "A DSP-Based Real-Time Image Processing System," *Proceedings of the 6th International Conference on Signal Processing Applications and Technology*, pp. 1042-1046, Boston, MA, Aug. 1995.

[2] T. Bapty, B. Abbott, "Portable Kernel for High-Level Synthesis of Complex DSP-Systems," *Proceedings of the International Conference on Signal Processing Applications and Technology*, Boston, MA, Aug, 1995.

[3] A. Misra, J. Sztipanovits, A. Underbrink, J. R. Carnes, B. Purves, "Diagnosability of Dynamical Systems," *Proceedings of the Third International Workshop on Principles of Diagnosis*, Rosario, WA, Oct. 1992.

[4] B. Abbott, T. Bapty, C. Biegl, G. Karsai, J. Sztipanovits, "Model-Based Approach for Software Synthesis," *IEEE Software*, pp. 42-53, May 1993.

[5] G. Karsai, J. Sztipanovits, S. Padalkar, F. DeCaria, "Model-embedded On-line Problem Solving Environment for Chemical Engineering," *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pp. 227-233, Ft. Lauderdale, FL, Nov. 1995.

[6] J. R. Davis, "Integrated Safety, Reliability, and Diagnostics of High Assurance, High Consequence Systems," *Ph.D. Dissertation*, Vanderbilt University, 2000.

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

[8] A. Misra, G. Karsai, J. Sztipanovits, A. Ledeczi, M. Moore, "A Model-Integrated Information System for Increasing Throughput in Discrete Manufacturing," *Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems*, pp. 203-210, Monterey, CA, Mar. 1997.

[9] M. Moore, S. Monemi, J. Wang, J. Marble, S. Jones, "Diagnostics and Integration in Electric Utilities," *Proceedings of the IEEE Rural Electric Power Conference*, Louisville, KY, May 2000.

[10] 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.

[11] J. Scott, S. Neema, T. Bapty. "Runtime Environment for Dynamically Reconfigurable Embedded Systems," *Proceedings of the International Conference on Signal Processing Applications and Technology*, Orlando, FL, Nov. 1999.

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

[13] *Using Matlab*, The MathWorks, Inc, 1999.

[14] A. Ledeczi, M. Maroti, G. Karsai, G. Nordstrom, "Metaprogrammable Toolkit for Model-Integrated Computing," *Proceedings of the Engineering of Computer Based Systems (ECBS) Conference*, pp. 311-317, Nashville, TN, March, 1999.

[15] J. Sztipanovits, G. Karsai, C. Biegl, T. Bapty, A. Ledeczi, A. Misra, "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pp. 361-368, Ft. Lauderdale, FL, Nov. 1995.

[16] B. Cmelik, D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proceedings of the Third Workshop on Computer Architecture Education*, Feb. 1997.

[17] G. Bassak, "Focus-Report: HDL Simulators," *Integrated System Design*, Jun. 1998.

[18] http://www.mathworks.com/products/simulink

[19] http://www.mathworks.com/products/xpctarget/index.shtml

[20] E. Lee and D. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, pp. 237-247, Jan. 1987.

[21] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentellli. "Hardware/software codesign of embedded systems," *IEEE Micro*, 14(4):26-36, August 1994.

ELECTRICAL ENGINEERING

INTEGRATING HIGH-LEVEL SIMULATION INTO A MODEL-INTEGRATED EMBEDDED SYSTEM DESIGN TOOLSET

BRANDON K. EAMES

Thesis under the direction of Dr. Gabor Karsai

The design of modern complex embedded systems is difficult. Resource, performance, and cost constraints require application-specific implementations. Design engineers often apply simulation techniques early in the design process to uncover problem areas prior to implementation, before they result in costly errors and design flaws. However, even with advances in simulation techniques, the design of complex systems remains a challenge.

Through the Adaptive Computing Systems (ACS) project at the Institute for Software Integrated Systems, a high-level system design tool has been developed to aid the design of adaptive embedded computer systems. The tool applies the principles of Model-Integrated Computing, allowing a designer to create high-level models of a system, and then directly generate system specifications and architectures from the models. The use of this tool greatly simplifies many of the complexities involved in embedded system design.

While the ACS design tool incorporates principles of model-integrated computing in the design of an actual system, it does not provide any support for simulation. This

thesis describes an extension of the ACS toolset to allow simulation techniques to be integrated into a Model-Integrated embedded system design process. Specifically, the Matlab language and computation environment are integrated into the design flow, allowing models to be functionally verified prior to implementation, and components to be implemented in the Matlab language and executed as part of a heterogeneous embedded system.

Approved_____ Date_____