Model-Integrated Environment for Adaptive Computing

J. Scott, T. Bapty, S. Neema, and J. Sztipanovits
Vanderbilt University / Institute for Software Integrated Systems
Department of Electrical and Computer Engineering,
Box 1826 Station B, Nashville, TN 37235

Abstract

Many high-performance, embedded applications must function in rapidly changing environments. Power/size constraints limit hardware size, while performance requirements demand algorithm-specific architectures. Reconfigurable computing devices allow the architecture to change in response to the changing environment.

A model-integrated approach is used for the synthesis of these systems. The target systems are built on a heterogeneous computing platform: including configurable hardware, ASIC and general-purpose processors and DSPs. The model interpretation process will generate hardware/software architecture specifications and a run-time Configuration Manger allowing dynamic adaptation to changing environments while the synthesized system is on-line. The synthesis process will optimize hardware/software architectures for user-definable cost functions such as weight, power, algorithmic accuracy and flexibility.

I. INTRODUCTION

This research is motivated by the requirements for design and implementation of adaptive missile automatic target recognition (ATR) systems. ATR systems have extremely large computational requirements. Image sizes are large with a high frame rate (30 frames/sec). Processing of this input data must meet hard real-time requirements. In order to achieve these requirements many components of this processing must be implemented in hardware; other components may be implemented in software.

ATR systems must be physically small, less than 1 cubic foot. Weight is also a major consideration. These factors require that component utilization be maximized as much as possible for selected hardware. ATR systems also require special attention to power consumption. During some processing modes components not needed to meet processing requirements at that time can be put into a low-power mode or shut down.

In the design process of many complex systems it is not obvious which implementation choices will yield a good balance between power usage, hardware size, and system performance. Also, design cycles for these systems tend to be long. By the time a design is completed the hardware platform that the system was designed for may become obsolete. An adaptive system is needed to support a hardware platform that may be evolved to include current state-of-the-art technology without complete redesign.

In a single mission, an ATR system traverses a large number of operation modes (target acquisition, target tracking, aim point selection, etc.). Each mode can have very different processing requirements (latency/throughput/accuracy), resources, and operational constraints. Different modes may have a different computational structure. Reconfigurable computing offers a method for maximizing the utility of the computational platform by adapting architectures to the changing needs of the system. This provides reuse of components over time.

The system hardware configuration our toolset supports may be constructed of both CPU processors and hardware processing elements. The processors may be DSPs (Digital Signal Processors) or conventional RISC/CISC processors that execute software processes. The hardware processing elements may be fixed-function devices such as an ASIC FFT implementation or a programmable logic device such as a FPGA (Field Programmable Gate Array). FPGAs have the ability to implement arbitrary hardware functions. FPGAs can also be reprogrammed quickly to completely change the function(s) implemented. Currently FPGAs are evolving quickly to have faster (re)configuration times and larger capacity. These properties make FPGAs a good choice for high-performance processing applications where flexibility is needed [1]. Multiple processing functions may be allocated to FPGAs and can change as the design evolves or the processing functions may be completely changed on the fly during system operation as processing requirements change.

The design of reconfigurable computing systems represents a significant challenge to the engineering process. System complexity skyrockets, since we are no longer designing a single system on a fixed architecture. We must now consider the design to be an integration of subsystems, each subsystem representing a phase of the system, with subsystems spread out over time. Additional complexities arise from the need for all subsystems to share the same physical implementation.

The Institute for Software Integrated Systems is developing an approach for managing the complexity in designing reconfigurable systems. Experience with Model-Integrated Computing (MIC) has shown itself to be successful in comparable situations. The MIC approach involves the following steps:

- Use **Multi-aspect, Domain-specific Modeling Environments** to capture requirements, design methods, and resources in a format that is customized to the problem and its formalisms.

- Develop **System Synthesis tools** for converting the models into executable artifacts.
- Develop a **Runtime Execution Environment** for supporting the execution of the generated system.

II. Modeling Concepts

A multi-aspect, domain-specific modeling paradigm was chosen to capture all information necessary to model and synthesize a system [2]. This information includes system computation requirements, computation algorithm(s), and available system resources. The modeling paradigm has three main aspects: a *structural aspect*, a *behavioral aspect*, and a *resource aspect*. Each of these aspects is defined in a graphical language customized for the domain of adaptive computing systems.

A. Structural Aspect

The structural modeling aspect is used to describe the processing algorithm structure. This structure may be defined hierarchically and may also include multiple algorithm architecture alternatives for a given task. The use of these two concepts together provides a powerful method for modeling the possible design space. This design description may describe a potentially huge number of design implementations. Modeling the different ways in which a processing task may be realized gives the design environment the freedom to search for and select an implementation that meets the specified requirements and fits within the resources available.

The algorithm is modeled as a dataflow structure with the following objects: *compounds*, *primitives*, and *templates*. The relationship between these objects is shown in Figure 1. A primitive is a basic element representing the lowest level of processing that is modeled. A primitive is a processing object that is to be implemented as either a hardware function or a software function. A compound is an object that may contain primitives, other compounds, and/or templates. Compounds provide the hierarchy in the structural description that is necessary for managing the complexity of large designs.

A *template* object is used to describe the concept of a choice between multiple design architectures. A template can be used to model different *algorithm* alternatives or different *implementation* alternatives. For example, many types of signal processing tasks can be accomplished in the spatial or the spectral domain. Both approaches may achieve the same basic results but with vastly different algorithm designs. In the spatial domain a filtering function may be achieved by performing a time domain convolution. In the frequency domain the function may be achieved by performing a FFT, followed by a multiplication, then an inverse FFT. It is useful to model multiple implementation alternatives, that is, different ways a processing function may be implemented. For example, a convolution can be computed in software running on a DSP, software running on N DSPs, hardware function in a FPGA, or a dedicated ASIC solution.

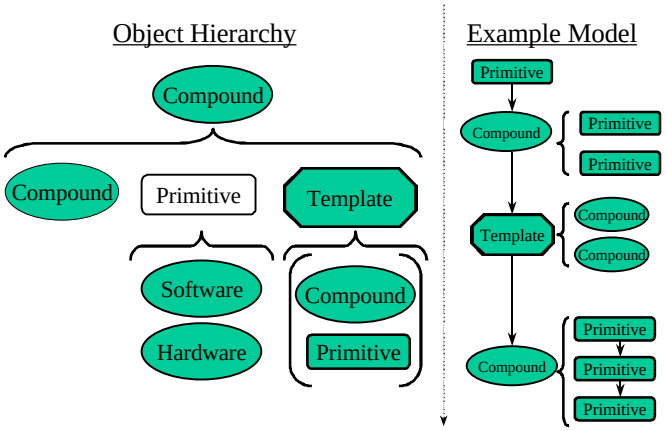

Figure 1: Object hierarchy definition of Structural Aspect along with an example instantiation

The use of these types of algorithm alternatives allows our model of the system to capture design possibilities. Each of these alternative methods has different performance attributes and different hardware requirements. Which alternative will be a good choice depends not only on the hardware that is available but also if the hardware is to be time-shared what hardware is required of the other processing algorithms that are operating in different modes. Although the type of hardware (FPGA, DSP, etc.) that basic processing primitives can be implemented on is part of their definition, the hardware resources that are available may not be known or may change at this point.

B. Behavioral Aspect

The behavioral aspect defines the *modes* in which the system may operate and the manner in which these mode changes can occur. Each mode is defined by the processing algorithm that is to be operational in that mode. The possible transitions from one mode to another are specified in the behavioral aspect by a state transition graph as shown in figure 2.

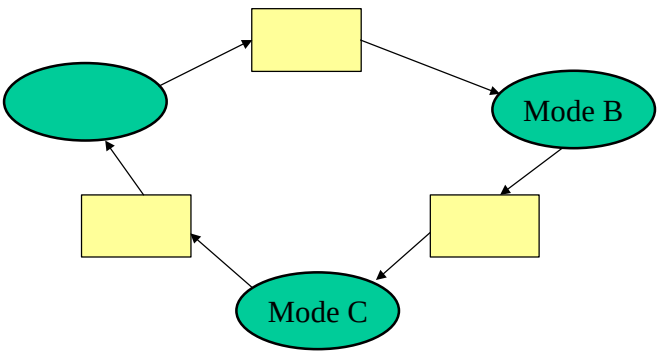

Figure 2: Behavioral Aspect

The event expression that can trigger a mode change is defined by the *transition rules*. A transition rule is a Boolean equation composed of event variables. When this expression is satisfied the transition from one mode to another is enabled and system reconfiguration is to take place.

C. Resource Aspect

The resource aspect defines the hardware platform available for the system to be implemented. The hardware platform is modeled in terms of hardware components and the connections among them. The relationships among the resource model components are shown in figure 3. The hardware system is a network of components that are either processor type elements such as DSPs or standard RISC/CISC processors, programmable logic components such as FPGAs, or dedicated hardware ASIC components for specific functions such as FFT computation. Items known as *cores* and *ports* are used to describe processing elements. A processing element must contain a core. The core object captures the necessary performance attributes of the processing element such as clock speed, contain a. A core represents the processing element A port represents a physical communication channel that a processing element has available. Connections between processing elements are created by connections between ports.

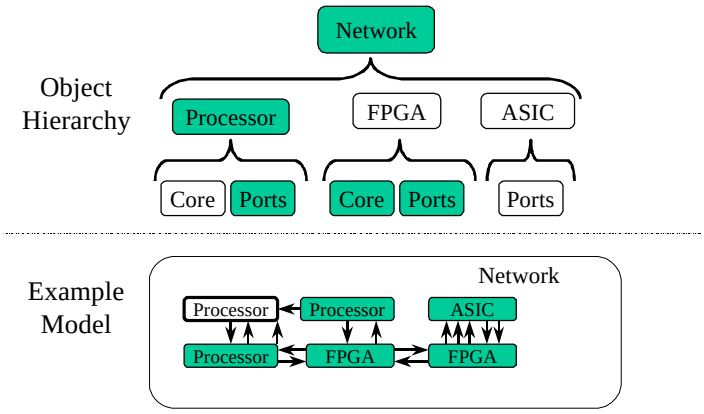

Figure 3: Resource Model

These three aspects of our modeling paradigm are integrated into a single design environment, known as GME (Graphical Model Editor).

III. Runtime Execution Environment

The runtime environment must support implementation platforms with the following attributes:

- Heterogeneous: Optimizing the architecture for performance, size, and power requires that the best implementation techniques be used. Implementations will require software (implemented on RISC and DSP processors), configurable hardware on FPGAs, and a mix of ASIC components.
- Performance: the runtime environment must minimize overhead, since overhead results in extra hardware requirements.
- Real-Time: The target systems have significant real-time constraints.
- Reconfiguration: The execution environment must allow hardware and software resources to be reallocated dynamically. During reconfiguration, the application data must remain consistent and real-time constraints must be satisfied.

In addition, the runtime environment must be designed with an interface suitable for synthesis from a MIC-Generator approach. The properties of the runtime environment must be tuned to simplify the generator. This demands a simple, uniform interface with a well-defined, consistent set of semantics that apply throughout the system.

The execution environment has been derived from tools developed over the past several years. These tools are used to construct large-scale, parallel, real-time signal processing systems. The runtime environment enabled development of CADDMAS systems, which are used by the USAF for turbine engine testing and NASA for SSME monitoring and analysis.

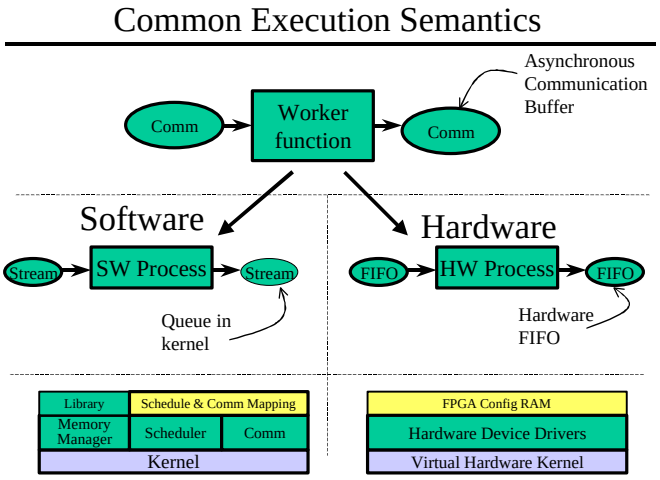

Figure 4: Execution Environment Semantics

The semantics of the execution environment implement a large-grain dataflow architecture. Processes and Processors are equivalent, representing functions on data. Processes/processors are connected via logical streams/signals that must buffer, communicate, and match data formats. Figure 4 illustrates the common execution semantics that exist between hardware and software platforms and the different implementations required to enforce these semantics.

The execution environment spans software and

reconfigurable hardware. The software environment consists of a simple, portable real-time kernel with a run-time-configurable process schedules, communication schedule, and memory management. Communications interfaces are supported within the kernel, making cross-processor connections invisible. Memory management is integrated with the scheduler and communication subsystems, enabling (but not solving) the problems associated with dynamic reconfiguration. The software kernel uses the communication infrastructure broadcast commands from the Reconfiguration Manager to receive configuration information.

The hardware execution environment is semantically similar, but the implementation is much different. The Virtual Hardware Kernel exists as a concept only. The MIC Generator synthesizes the processors, the signal buffering, and the necessary off-chip interfaces and data converters. These interfaces generated at the hardware/software boundaries provide a practical means to produce a synthesizable system and facilitate the HW/SW codesign process [3].

IV. System Synthesis

The multi-aspect model of the system describes a possibly enormous number of design solutions. The set design solutions must be evaluated to find a set of designs (mode configurations) that best satisfy a number of design criteria. This is a very difficult task because there are inherently a large number of conflicting design criteria in reconfigurable systems. Each mode has performance requirements that demand a certain level of performance from the hardware for a given algorithm. Some hardware components are more suitable for certain tasks than others. DSPs provide a general-purpose solution and reasonable performance for many complex algorithms while ASICs can provide a high-performance solution at the cost of adding a dedicated fixed-function IC. The processing needs of the multiple modes of the system must be met for a single shared hardware platform. The synthesis process will select feasible hardware/software architectures for user-definable cost functions such as weight, power, algorithmic accuracy and flexibility.

The structural aspect that captures the hierarchical data-flow with alternatives can represent an exponentially large design space in a compact form. An algorithm structure may have an extremely large number of possible implementations but a very small solution space as visualized in figure 5.

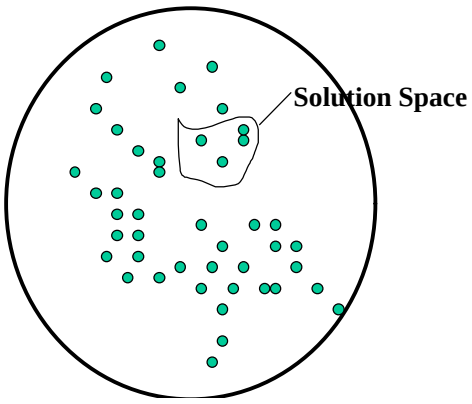

Figure 5: Design Space Visualization

Searching the design space to a set of configurations to satisfy the design criteria is accomplished in multiple stages, each with increased resolution. A symbolic constraint satisfaction method is introduced to provide an initial pruning of the design space. This method operates on the binary design and performance constraints specified for the systems. This method of pruning is implemented using a symbolic method known as ordered binary decision diagrams (OBDD). A symbolic representation is built along with a constraint set. System design alternatives that do not satisfy this constraint set may be quickly eliminated. Through the use of the OBDD representation these design possibilities do not have to be examined individually allowing for an extremely large design space to be quickly narrowed.

The design search will continue to narrow down possibilities through high-level simulation of the system. Components will have associated performance models that can be used to compute performance data of the system configuration being evaluated such as communication utilization, processor utilization, etc. Finally when the searching process has narrowed the design space down to only a few candidate configurations it may be necessary to fully synthesize these configurations for the target platform and evaluate their performance through actual run-time measurements.

At this point, the synthesis procedure can generate the actual runtime artifacts. From the behavioral models, a set of tables is produced for the Configuration Manager. This defines the behavior of the system, in terms of a state machine. These tables are executed directly by the configuration manager.

For each configurable component, a set of design files is generated. One file is built for each component for each mode. The design is specified in structural VHDL, using computational components from the design library and interface components from the runtime system library. These VHDL files are then compiled using vendor-supplied/COTS VHDL compilers and part-specific Place-and-Route tools. The result is a “bitfile”, ready for direct device programming.

For the general-purpose/DSP components, a set of real-time schedule specifications and communication maps are generated. These are then processed into a set of object modules and tables for direct download into the parallel array of processors. One set of tables is built for each processor for each mode, and one common executable module is generated for each processor.

The result of the synthesis and post processing is a complete executable system, ready for deployment. Current synthesis capabilities do little optimization (Optimization is an ongoing research topic).

CONCLUSIONS

Certain high-performance, highly-constrained applications need to be adaptable to their requirements and environment. Needed is a method for rapid, automated system synthesis that can provide maximal use of available hardware over time through system reconfiguration. FPGAs are an enabling technology for a computing platform that is able to adapt to changes in the processing algorithm. FPGAs may be used as general purpose computing devices whose flexibility and speed fall somewhere between that of a custom ASIC and that of a standard CPU.

The use of a domain-specific, multi-aspect modeling paradigm is key to capturing all relevant information about the system in an integrated environment. With this information design choices can be automated to select from the possible configurations a system configuration that meets specified design requirements and is also optimized for other specified metrics such as power use, weight, and algorithmic accuracy. Extremely large design spaces can result from the freedom given in defining the algorithm structure alternatives. Methods are being investigated to manage these large state spaces symbolically using Ordered Binary Decision Diagrams (OBDDs). Symbolic manipulation can provide a way to prune the design space without examining each design alternative individually.

Dynamic reconfiguration provides a better utilization of hardware over the different operational modes of the system. Automatic target recognition can benefit greatly from dynamic reconfiguration because of nature the ATR application; each phase has a different computational structure. The ability to reconfigure the processing structure to fit the current processing structure can minimize redundant hardware.

The current modeling environment is being evaluated. Applications are being modeled to ensure all relevant information is captured in the models. A set of intrinsic components is being constructed for the hardware communication interfaces. Dynamic reconfiguration approaches are being tested and refined. Applications are being built to evaluate and refine components and approaches.

Many issues need to be explored with respect to dynamic reconfiguration. Suppose processing algorithm A is operating and reconfiguration occurs to switch over to algorithm B, what does an output common to both of these algorithms produce? Is a discontinuity at the point of reconfiguration observed? If these outputs are used to provide information to a control system this may be unacceptable. A possible solution could be to define (model) a transition phase between the mode changes to provide a more continuous transition between modes. The issue becomes more complicated if the system is allowed to be partially reconfigured.

The systems currently being explored are real-time embedded systems. Reconfiguration must be real-time in nature to meet deadlines.

ACKNOWLEDGMENTS

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

REFERENCES

[1] J. Villasenor, W. H. Mangione-Smith, "Configurable Computing," *Scientific American*, pp.66-71, June 1997.

[2] J. Sztipanovits, G. Karsai, T. Bapty: “Self-Adaptive Software for Signal Processing,” *CACM*, Vol. 41, No. 5., pp. 66-73, May, 1998.

[3] P. Mertens, "System Architecture Design Using Structured Methods," *Codesign Computer-Aided Software/Hardware Engineering*, Chapter 12, pp.299-323, IEEE Press, Piscataway, NJ, 1995.