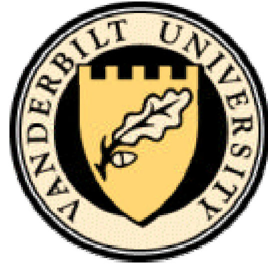
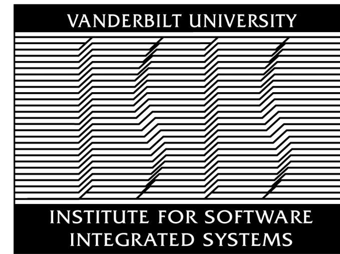




Institute
for
Software-Integrated Systems

Technical Report

TR #: **ISIS-99-01**

Title: **Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems**

Authors: **Ted Bapty, Sandeep Neema, Jason Scott, Janos Sztipanovits, Sameh Asaad**

Abstract

Several classes of modern applications are demanding very high performance from systems with minimal resources. These applications must also be flexible to operate in a rapidly changing environment. High performance with limited resources needs application-specific architectures, while flexibility requires adaptation capabilities. Reconfigurable computing devices promise to meet both needs. While these devices are currently available, the issue of how to design these systems is unresolved.

This paper describes an environment for design capture, analysis and synthesis of dynamically adaptive computing applications. The representation methodology is captured in a Domain-Specific, Model-Integrated Computing framework. Formal analysis tools are integrated into the design flow to analyze the design space to produce a constrained set of solutions. HW/SW Co-simulations verify the function of the system prior to implementation. Finally, a set of hardware and software subsystems are synthesized to implement the multi-modal, dynamically adaptive application. The application executes under a runtime environment, which supports common execution semantics across software and hardware. An application example is presented.

KEYWORDS

Reconfigurable Computing, FPGA, HW/SW Co-design, HW/SW Synthesis, FPGA, HW/SW Co-simulation, Dynamic Reconfiguration, Design Environment, Model-Integrated Computing.

ACKNOWLEDGEMENTS

This work has been supported by DARPA/ITO under project **DABT63-97-C-0020**

INTRODUCTION

Modern high-performance embedded systems, such as Automatic Target Recognition for Missiles or Dynamic Protocols Mobile Communications devices, face many challenges. Power and volume constraints limit hardware size. Accurate, high-performance algorithms involve massive computations. Systems must respond to demanding real-time specifications. In the past, custom application-specific architectures have been used to satisfy these demands.

This implementation approach, while effective, is expensive and relatively inflexible. As the world demands flexible, agile systems, the hardwired application-specific architectures fail to meet requirements and become expensive to evolve and maintain. As new algorithms are developed and new hardware components become available, a fixed, application specific architecture will require significant redesign to assimilate the technologies.

Flexible systems must function in rapidly changing environments, resulting in multiple modes of operation. On the other hand, efficient hardware architectures must match algorithms to maximize performance and minimize resources. Structurally adaptive, reconfigurable architectures can meet both these needs, achieving high performance with changing algorithms. Reconfigurable computing devices, such as Field Programmable Gate Arrays allow the implementation of architectures that change in response to the changing environment.

The field of Reconfigurable Computing is rapidly advancing for scientific and Digital Signal Processing applications [1][2][3]. While today's Field Programmable Gate Array technology shows great promise for implementing reconfigurable computational systems, their capabilities in certain areas (such as floating point arithmetic) cannot equal other technologies. For this reason, efficient system architectures must encompass a heterogeneous mix of the best technologies. The target systems are built on a heterogeneous computing platform: including configurable hardware, ASIC and general-purpose processors and DSPs.

The primary difficulty in this approach lies in system design. A designer must now maintain a set of diverse system architectures, which exist at different times in the system's lifetime, and map these architectures onto the same group of resources. The designers must manage the behavior of the system, determining the operational modes of the system, the rules for transitioning between operational modes, and the functional properties within each operational mode. In addition, the system must make efficient use of the resources, enabling the designer to minimize the envelope of hardware required to support the union of all operational modes. Current system design tools are insufficient to manage this complexity.

High-level design tools are being developed to capture designs and to generate functional systems as part of the DARPA Adaptive Computing Systems Program. This paper describes a *model-integrated* approach to be used in the development of reconfigurable systems. There are many significant issues in the development process. The approach described here divides these issues into several categories: (1) Representation and Capture of design information in terms of *Models*; (2) Analysis of the models for design/requirements/resource trade-off studies; (3) Synthesis of architectures and executable systems directly from the models; and (4) Runtime support environments to support efficient execution of the synthesized reconfigurable systems.

The Model-Integrated Computing (MIC) approach has been successfully applied to a diverse set of applications ([4][5][6][7][8][9]). The general MIC approach involves creating a development environment that is customized for a specific application domain. The resultant development environment is a *multiple-aspect* graphical editor that directly supports the engineering concepts required in the development process. Where several engineering disciplines are involved in system development (e.g. Software, Hardware, DSP algorithms, Systems Requirement Specification, etc.), the multiple-aspect nature of the approach allows different aspects to be customized for individual disciplines. The graphical editor allows construction of system Models, which capture the specifications and components required along with their relationships. The Models form a database of design information that can then be used in system analysis, trade-off studies, and performance estimation/simulation. These same Models are used to synthesize the executing systems. The synthesis process assumes a runtime environment that hides the low-level hardware/software details from the synthesis process..

This paper attempts a logical progression in describing the Model-Integrated Computing approach for adaptive systems design. The first section will describe the rationale and implementation of the design capture approach. The next section will give an overview of the current and planned analysis capabilities for design-space exploration. The following sections will describe the system synthesis process and the runtime environment architecture and implementation. Finally, we will show the implementation of a missile Automatic Target Recognition application incorporating adaptive system behavior.

DESIGN REPRESENTATION

The customization of the Model-Integrated Computing design environment involves a careful analysis of the needs of the design engineers, the methods and components used in the designs, and the target systems. For an environment to successfully support the creation of systems, the concepts used by designers must be faithfully reproduced by the design environment. This section will describe the concepts developed in the creation of the Adaptive Computing Systems MIC environment.

The Adaptive Computing Systems (ACS) environment divides the design process into four major categories:

1. **Behavioral Modeling:** In this first category, the operational adaptive behavior is defined. The designer can specify the operating modes of the system, the legal transitions between modes (and the conditions for transition), and the specifications for system operation while in each operating mode.
2. **Algorithm/Structural Modeling**: In this category, potential algorithms are described. The algorithms define signal flow specifications to compute required system outputs.
3. **Resource Modeling**: The resource models describe the hardware available for construction of the system. This consists of physical processors, devices, and the interconnection topology.
4. **Constraint Specification**: These modeling categories are augmented and linked together with a Constraint framework. The Constraints allow user-defined interactions to be specified, establishing linkages between properties in one category and objects in the same or another category.

Behavioral Models

Behavioral models capture the operational modes of the system and the potential interactions between these modes (Figure DR1). Since the system will be operating in discrete modes, with specific transitions between these modes, a familiar, well understood representation was chosen. The representation is a *Discrete Finite State Machine*.[10] *State*s define operational modes of the system. Transitions define the potential conditions required for the system to change modes and the end-state of the mode-shift. In order to manage system complexity, where the system may have many potential operational modes, a Hierarchical description was chosen.

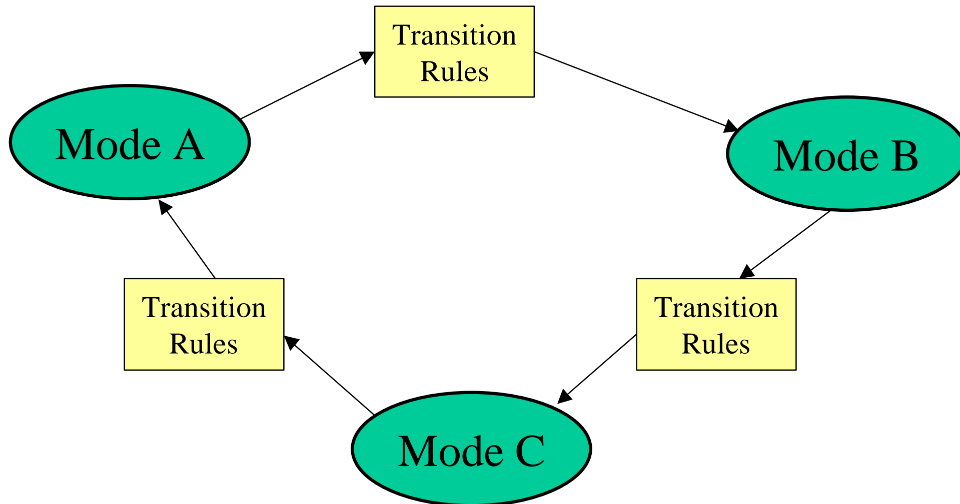


Figure DR1: Behavioral Model

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. The event variables are computed in the Algorithmic/Structural modeling view described below. These can be directly sampled external signals or complex computational results.

The behavioral modeling aspect is linked to the Algorithmic/Structural aspect by the means of *References*. A Reference is a modeling "trick" that allows the user to establish a pointer from the mode to a defined computational algorithm. Each mode references a model in the Structural Aspect that defines the processing algorithm that is to be operational in that mode. The references allow a single algorithm to be applied to any number of system states, or allow all states to have separate processing structures.

The behavioral modeling aspect also allows the specification of real-time requirements and maximal runtime power usage. Maximal permitted system delays can be specified for any pair of input and output ports on the structural model. The power characteristics are specified using attributes of the models, in which the designer can enter a maximum allowable power limit. In effect, the Behavioral Models capture the system performance requirements.

Algorithm/Structural Models

The structural modeling aspect is used to describe the processing algorithm structure. The basic algorithm is described in terms of computational components and data interactions. To manage system complexity, the concept of hierarchy is used to structure algorithm definition. This logical composition of systems using component subsystems has proven effective design structuring for very large, complex systems.

The algorithm is modeled as a dataflow structure with the following classes of objects: *compounds*, *primitives*, and *templates*. The relationship between these objects is shown in Figure DR2. A primitive is a basic element representing the lowest level of processing that is modeled. A primitive maps directly to a processing object that will be implemented as either a hardware function or a software function. Primitive objects are annotated with attributes. These attributes capture measured performance, resource (memory) requirements, and other user-defined properties.

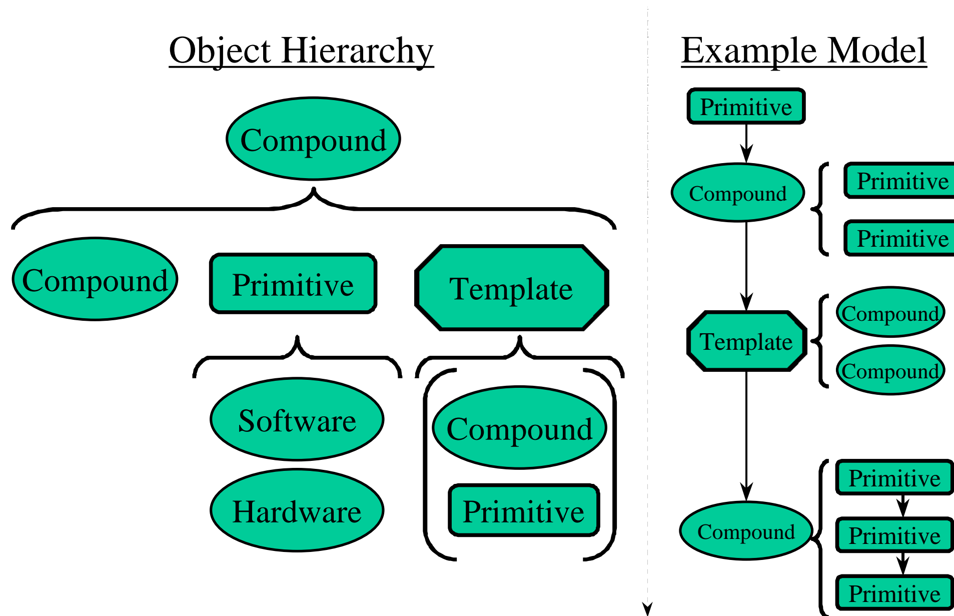


Figure DR2: Compound/Primitive/Template structure

A compound is an aggregation object that may contain primitives, other compounds, and/or templates. These components can be connected within the compound to define the information dataflow. Compounds provide the hierarchy in the structural description that is necessary for managing the complexity of large designs.

A design alternative object is used in the modeling process to allow the specification of multiple algorithm architecture alternatives for a given task. The *Template/Alternative* object is used to capture the design alternatives. This object represents a choice between multiple design architectures. These design alternatives can be either Compounds or Primitives, allowing hierarchies of design alternatives.

When alternatives are used, the algorithm structural models describe a huge number of potential design implementations. The large design space gives environment the freedom to search for and select an implementation that meets the specified requirements and fits within available resources.

In signal processing, many types of tasks can be accomplished in multiple ways, for

example in the spatial or the spectral domain. Both approaches will achieve the same basic results but with vastly different algorithm designs. Other algorithm characteristics can vary as well, such as latency and/or accuracy. In the spatial domain a filtering function can be achieved by performing a standard mathematical convolution. In the frequency domain, the function is achieved by performing a FFT, followed by a multiplication with the spectral representation of the filter, followed by an inverse FFT. In this case, the spectral method is more efficient as the filter order increases, resulting in a faster, smaller system. On the other hand, since the FFT is a block-based computation, the latency is at least a block-length.

Algorithm alternatives allow the model of the system to capture design possibilities. Each of these alternative methods has different performance attributes and different hardware requirements. The selection of the best alternative depends not only on the hardware that is available, but also on whether the hardware is to be time-shared, and what hardware is already allocated to support the processing algorithms that are required for operations in different modes.

For the high-level designer, algorithm alternatives allow a virtual separation of algorithm from implementation. Typical algorithm design requires the engineer/physicist to consider the hardware details of the underlying architecture to achieve an efficient implementation. The ultimate effect is that the resulting algorithm reflects the hardware structure. This practice leads to highly non-portable, technology-specific designs. System upgrades to use more modern technology require a bottom-to-top redesign. Algorithm alternatives promise to separate the algorithm from the architecture, to postpone the implementation decisions to a much later step in the design process. This approach should greatly simplify technology migration efforts.

Another use of templates is to model multiple physical technology implementation alternatives, i.e. different ways a processing function may be implemented in the architecture. For example, a convolution can be computed in software running on a DSP, in software running on a network of multiple DSP's, in a hardware function in a FPGA, or in a dedicated ASIC solution. The selection of the desired implementation technology is determined in the synthesis process, driven by power consumption, throughput, latency, specific part availability, and other architectural interactions.

Resource Models

The resource aspect defines the hardware platform available for the target application. The target hardware platform is modeled in terms of hardware components and the physical connections among them. The relationships among the resource model components are shown in figure DR3.

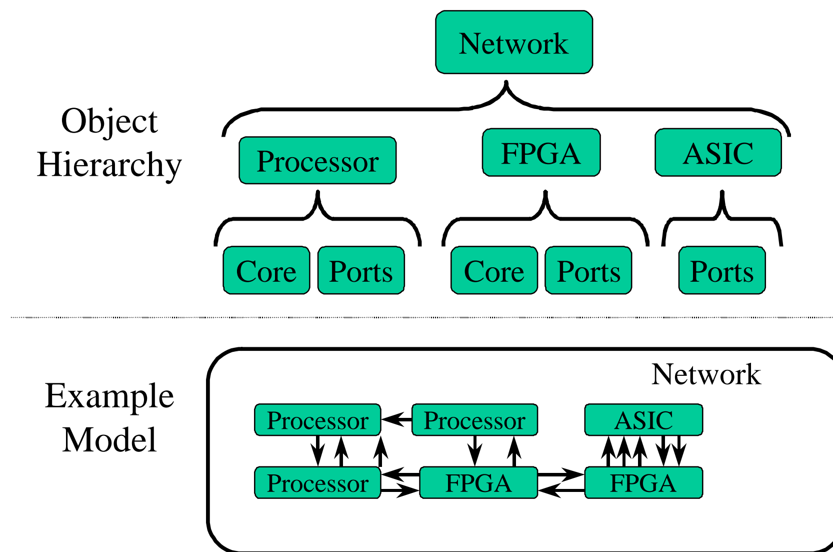


Figure DR3: Resource Model

The top-level hardware system is a *Network* of components. Network components are either processor elements (such as DSPs or standard RISC/CISC processors), programmable logic components (such as FPGAs), or dedicated hardware ASIC components for fixed functions (such as FFT computation). Data Sources and Data Sinks capture the specifics of hardware I/O interfaces and data acquisition/effector interfaces.

The components are constructed using *cores* and *ports.* Every processing element must contain one core. The core object captures the inherent performance attributes of the processing element such as clock speed, memory, and other resources. (The core represents the processing element). A port represents a physical communication channel. Ports have associated protocols and specific pin assignments, representing physical connection points on a chip. Connections between processing elements are created by connections between ports. The connections capture the "as-built" topology of the physical implementation.

Constraint Specification

System constraint specifications have four categories of design constraints: (a) operational constraints, (b) composability constraints, (c) resource constraints, and (d) performance constraints to establishing linkages between properties in different modeling categories.

Operational constraints express conditions relating design configurations to operational modes. These constraints are applied within the Behavioral models. Composability constraints are logic expressions that restrict the composition of alternative processing blocks (eg. FFT-HW can be used with IFFT-HW). Resource constraints are logic

expressions describing the selection of processing blocks based on resource limitations. Performance constraints are integer constraint expressions limiting the end-to-end latency, power and space. The performance constraints are implicit in the properties of Behavioral models. These constraints allow the designer to control the potential design space for the analysis/synthesis process.

MODEL ANALYSIS

The end-product of the design process described above is a design space consisting of modes & requirements, potential implementations, and resource sets. The task of the designer is to select appropriate combinations of implementations and resource assignments for all of the desired operational modes. Given the flexibility in defining design alternatives, this space can be extremely large (moderately sized design examples have defined a space of 10^{24th}). It is unreasonable to assume that a designer can handle such a large design space without sufficient tools. The set of design solutions must be evaluated to find a set of designs (mode configurations) that best satisfy a number of design criteria. 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. The processing needs of each of the system modes must be satisfied with a single shared hardware platform. The analysis tools must allow efficient exploration, navigation, and pruning of this space to select feasible hardware/software architectures for user-definable cost functions such as weight, power, algorithmic accuracy and flexibility. Given the size of the design space, and the complexity of the analysis, a powerful analytical method is required.

Constraint Satisfaction using Symbolic Methods

The approach we have taken is to use symbolic methods based on Ordered Binary Decision Diagrams to represent, navigate and prune the design space. In a symbolic representation, sets/spaces are represented as a boolean expression over the members of the set. The members of the set are encoded as binary variables under a binary encoding scheme. The principal benefit of the approach is that it does not require enumeration of the set/space to perform operations. Ordered Binary Decision Diagrams [11][12] are a canonical representation of logic functions, representing boolean functions as directed acyclic graph in a memory-efficient format. The operations over the boolean functions are implemented as graph algorithms rendering “manipulation” of the space fast and efficient.

With this symbolic formalism, the application of logical constraints is relatively straightforward. The user-defined logical constraints can be represented as a boolean expression over the components of the design space. . Constraint application is then just conjunction of the constraint boolean expression with the boolean expression that represents the design space. The resultant boolean expression represents the “constrained” design space. Application of the integer arithmetic constraints such as

timing and power constraints is not so straightforward (see [15] for details). However the basic approach remains the same.

The constraints “prune” the design space due to the requirements specified in the constraint. These constraints can be iteratively applied to the design space, with the goal of reducing the “10^{24th}” to a more manageable 10-1000 design alternatives. We have implemented the approach described above in a design space management tool (Figure MA1) that allows solving these constraints in an iterative manner. The design engineers can apply the constraints and visualize the sensitivity of the design space to the constraint. If the constraint is extremely tight it can be released and other constraints can be applied instead. Finally when the design engineer is satisfied with the remaining design choices after constraining the design space he can move to the next step of simulation.

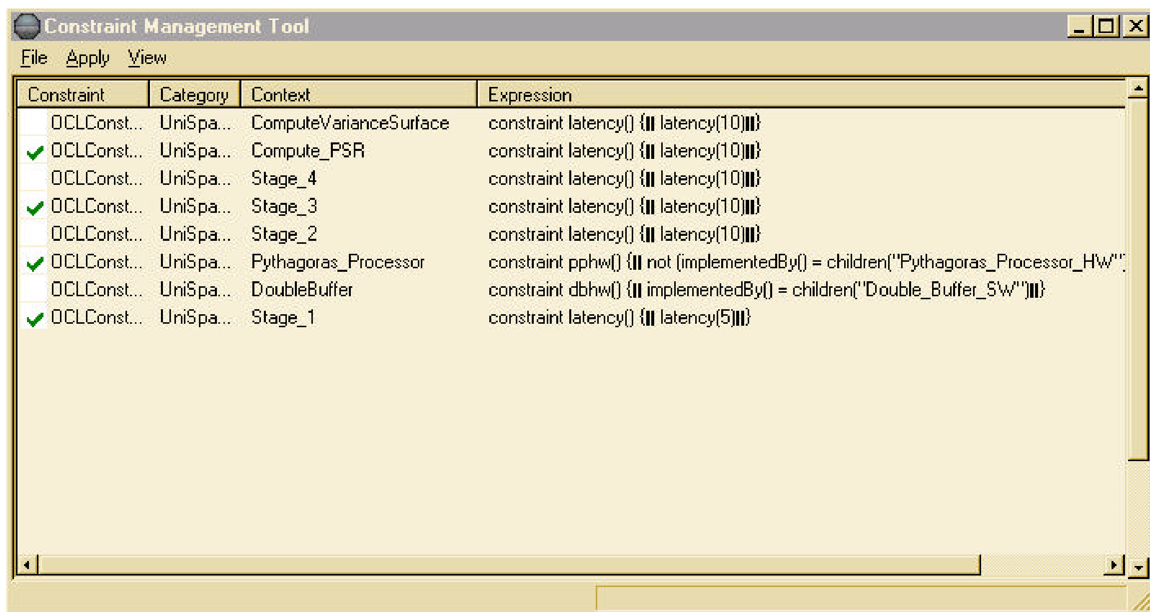


FIGURE MA1: CONSTRAINT MANAGEMENT TOOL

HW/SW Co-Simulation

The constraints encode the behavior of the system with a relatively high level of granularity. While this is necessary to work within the tremendously large design spaces, the accuracy of this approach will be poor. The designer will be required to “give the benefit of the doubt” to designs that are near the fringes of the constraint envelope. To establish a more accurate estimate of in-system performance, a simulator is required. Since the target of the tool is hardware and software, the simulator must support co-simulation.

While this research is still at its early phases, the current approach is to allow the system designer to perform co-simulation at three levels of abstraction for trade-off between execution speed and accuracy of results, namely the *performance level*, the *algorithm level* and the *gate/instruction level*. This will enable the designer to quickly “zoom-in” on

the more viable design alternatives and perform more accurate simulations only on this subset.

An important aspect of the co-simulation environment is the seamless integration with the rest of the system. Information used to automatically construct the simulation testbench at various levels is directly extracted from the model database to ensure consistency among various levels of detail. Different levels of simulations will utilize different, possibly overlapping subsets of the model database. On the other hand, output from the simulation is interpreted and fed back in a high level form to the user in the same design environment.

At the performance level, only the performance of the structural model is simulated. In other words, performance attributes, such as latency and throughput, associated with processing primitives are used to construct a network of delay models for the system. Data flow is abstracted out at this level and represented by tokens for faster simulation via packages such as PML[13][16]. No distinction of hardware versus software implementation is made at this level, except for the relatively longer delays associated with software realizations. The output of this step will be an overall performance assessment of the proposed algorithm as well as flagging the critical components or *hot spots* of the system.

At the algorithm level, the functional computation itself is simulated but without low-level timing details so that the user can quickly verify the correct functionality of the system. Hardware functions are described in VHDL and software functions are described in C and encapsulated in a VHDL-wrapper entity. A commercial VHDL simulator equipped with a foreign language interface will be the target for mapping.

The lowest level of abstraction is the gate/instruction level co-simulation. At this level, a HW/SW co-simulation environment is constructed that models the system platform as described in the resource models of section 1. VHDL simulation models will be used to describe hardware components such as ASICs and FPGAs. Processor models can range from full functional models that mimic the internal architecture of the processor to simple bus functional models that only describe the interaction of the processors with external components but do not mimic the internal architecture [17]. The former is usually too expensive in terms of execution speed and also difficult to construct from scratch for complex processors. The latter approach is more suitable for debugging the hardware portion but not well suited for viewing software execution. An intermediate approach is to use an instruction set simulator (ISS) coupled with a bus functional model (BFM) to model the processor, such as described in [18]. The ISS will be used to simulate software execution while the BFM will mimic the interaction with the external circuitry. Synchronization techniques between the ISS and the BFM are needed to keep the simulation realistic.

SYSTEM SYNTHESIS

The tools capture system requirements, design information and alternatives, and the resources available for system implementation in the form of *Models*. The constraints developed during the Model Analysis phase, when applied to the design space, define a manageable set of implementation alternatives. Expected performance is estimated using the Co-Simulation tools, providing further assurance that the system will function to design specifications. The selected design alternatives must now be transformed to software and hardware for system implementation. We refer to this process as model interpretation.

A model interpretation process generates hardware architecture specifications, software modules, process/schedule tables, communications maps, synthesizable hardware specifications, and a run-time Configuration Manger for dynamic adaptation to changing environments. The synthesis process attempts to optimize hardware/software architectures for user-definable cost functions such as weight, power, algorithmic accuracy and flexibility.

The first phase in the optimization process is the successive application of incrementally tighter design constraints. The symbolic constraint satisfaction method described in the Design Analysis section is used to provide an initial pruning of the design space.

The design search will continue to narrow down possibilities through multi-resolution 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 the searching process has narrowed the design space down to only a few candidate configurations per system operational mode.

Configuration Manager Synthesis

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. The state-based behavior is defined in the Behavior Models. These models are transformed into a compact state table. The table contains next state equations for each operational mode. The interfaces to internal and external events are generated to provide the state transition variables to the state machine. These tables and variable interfaces are executed directly by the configuration manager.

Hardware Synthesis

For each configurable component (FPGA), a design specification is generated. This design specification includes a hardware design file for each component for each mode.

The design for a component*mode is specified in structural VHDL. The VHDL design incorporates computational components from the design library, which can contain user-defined VHDL behavioral descriptions and vendor-supplied Intellectual Property (IP) modules. These modules are glued together using components from a standard interface runtime library, which is part of the Runtime Environment described later. These interfaces connect computational components on the same chip with simple FIFO's and asynchronous handshaking interfaces. When the communication must occur across chip boundaries, or to software components, a set of more complex interface components are used. These interface components manage the physical hardware resources (pins and wires), buffer data, and multiplex multiple logical communications across a single set of wires. Where required, data format conversions are supplied.

These VHDL files are then compiled using vendor-supplied/COTS VHDL compilers and part-specific Place-and-Route tools. The result is a set of "bitfiles". One bitfile is generated for each reconfigurable hardware device for each mode. Given the current state of the FPGA market, demand has not yet forced the vendors to provide partially reconfigurable devices and support tools. For this reason, we treat each FPGA as an atomic part, configurable only with a full device reset. The approach proposed here will work for partial reconfigurable devices by treating a single device as multiple logical devices. In order for this to work, the vendor tools must provide methods for floor planning to restrict logical design components (i.e. all components within a single mode) to non-overlapping, regions that coincide with legal chip reconfiguration boundaries.

Software Synthesis

For the general-purpose RISC/DSP components, a set of software specifications is generated. These specifications provide the information needed by the Runtime Environment to enact the desired computational behavior. The Runtime Environment requires several categories of design files:

- Software Load Modules contain executable modules that are downloaded to the processors in the system. The system can generate a common load module that contains the superset of all executable functions (if memory is sufficient) or it will generate a customized module for each of the processors in the system. The customized module is clearly more memory-efficient.
- Real-time schedules contain the list of processes and their priorities. A unique schedule is generated for each processor and for each mode of operation.
- Communication maps describe the information flow between processes. These "streams" can perform communication between two modules on the same processor, or they can transport data across the network, through intermediate processors, and to a remote process anywhere in the system.

Interfaces between software modules and hardware modules/data sources/sinks are automatically inserted during the synthesis process. These interfaces perform the "care-and-feeding" of hardware interfaces, converting complex communication protocols into simpler hardware compatible protocols. The interfaces also multiplex multiple logical streams over a single physical port and perform data conversion functions.

These design files are processed into a set of object modules and tables for inclusion in the configuration manager and for direct download into the parallel array of processors.

The result of the synthesis and post processing is a complete executable system, ready for deployment. The deployment is performed in concert with the Runtime Environment.

RUNTIME ENVIRONMENT

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

- Heterogeneity: Optimizing the architecture for performance, size, and power requires that the most appropriate implementation techniques be used. Implementations will require software (implemented on RISC and DSP processors), configurable hardware on FPGAs, and a mix of ASIC components.
- Low Overhead/High Performance: the runtime environment must minimize overhead, since overhead results in extra hardware requirements.
- Hard 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.

These issues must be addressed at multiple levels. At the lowest level, the hardware must be capable of reconfiguration. Software-programmable components, such as DSP's and RISC processors, have excellent inherent hardware support for reconfiguration, since software has the ability to change system function by changing memory contents. Internal CPU hardware structures are designed to restrict dangerous conditions that could damage hardware. FPGA's are an unrestricted collection of gates, switches, and connectors. The safeguards built into CPU's do not exist and must be enforced manually. This protection must be provided by a cooperation of the design process and the runtime infrastructure.

At a slightly higher level, the internal state of software must be managed under changing tasking. Modern operating systems have evolved to support the flexible implementation of multiple tasks, with dynamic addition and removal of tasks on a single processor in the form of time-sharing and/or multitasking, and Real-time kernels allow time critical tasks to be dynamically scheduled on a single processor. These kernels typically do not address the consistency of dynamic reconfiguration for distributed networks of tasks.
Finally the issues of application-specific requirements must be addressed, to allow the peculiar requirements of specific numerical performance and timing to be achieved in an implementation. Potential solutions to these issues with consistency are addressed in the next section.

Hardware/System Consistency

The runtime system must avoid operational defects during a reconfiguration event. Hardware consistency can have many negative effects, from temporary loss of performance in an operational mode to hardware damage and total, permanent system malfunction. Typically, these deal with specific issues involving interfaces between hardware processes and/or devices. Some of these defects are illustrated in figure RE1.

Hardware Consistency After Reconfig

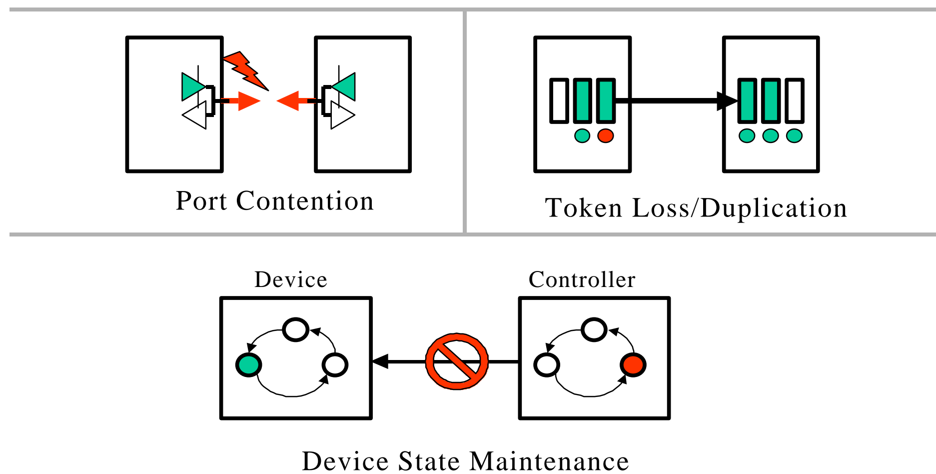


Figure RE1: Hardware Reconfiguration Problems: Maintaining Consistency

Port contention occurs when bi-directional ports are improperly initialized, a reconfiguration event is not properly sequenced/synchronized, or if an improper/inconsistent design is implemented. In this case, two connected drivers are enabled. If resistance is sufficiently low, permanent physical damage can occur to the circuits.

Token loss or duplication results from incorrect initialization or a loss of communication integrity. Tokens represent the status of empty or full slots in a communication interface. An extra token on the sender side can cause too much data to be sent, resulting in a FIFO overrun. A lost token can effectively block a communication port, resulting in a system deadlock.

Device state maintenance refers to the control of a complex external hardware device, such as an attached processor or storage device. In controlling an external device, the controlling computational component must maintain an accurate representation of the device's state. If a reconfiguration occurs during a state transition within the device, or if the reconfiguration modifies the computational component's representation of the device, there can be a state mismatch. This can result in improper commands being sent to the device, or in a deadlock where both components are waiting on each other for triggering events.

These three examples show some of the potential hazards that can occur when the hardware device is improperly reconfigured. Runtime reconfiguration support must not permit any of these conditions to occur.

Software/OS Consistency

Software issues can present a larger challenge to dynamic system reconfiguration. While the hardware built into standard microprocessor devices protects against low-level hardware conflicts, there are many more details that must be managed. Figure RE2 below summarizes some of the potential problems from an improper reconfiguration.

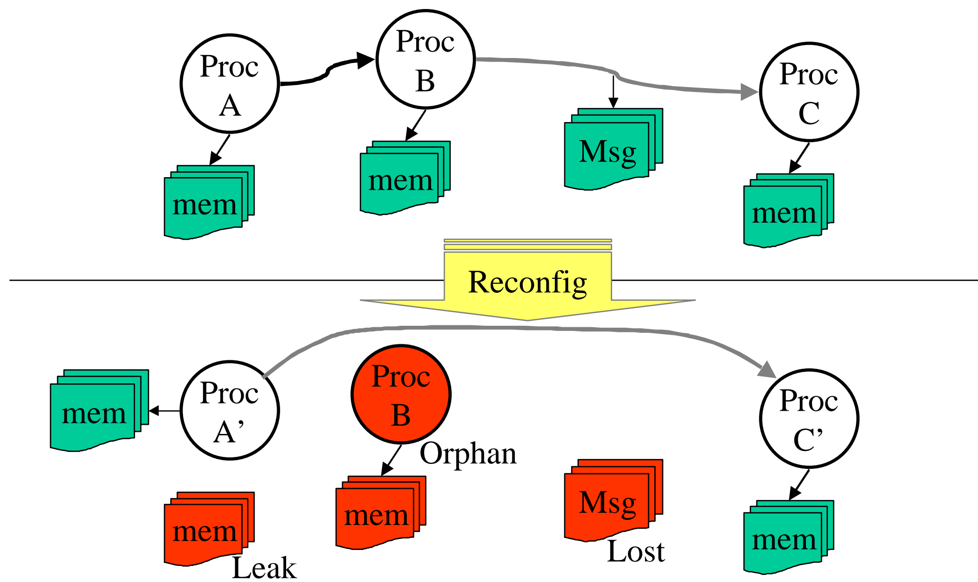


Figure RE2: Software/OS Reconfiguration Problems: Maintaining Consistency
The example shows an initial configuration of 3 processes (A, B, and C) in the normal operational state. A reconfiguration occurs, changing to a new configuration. The new configuration replaces these process A with A', C with C' and removes Process B altogether. The bottom half of the figure shows the new configuration, along with the potential errors.

Memory leaks will adversely affect long-term reliability. Task structure mismanagement results in extra tasks executed by the kernel, with a loss in performance. Messages in transit can be delivered when the receiving process no longer exists, resulting in mis-matched messages and channel errors.

Application-Level Consistency

At a higher level, the application's requirements and implementation details impose

restrictions in the reconfiguration process. Typically, these attributes are highly application-specific. Two examples of consistency requirements are displayed in Figure RE3 below.

1. An external system may require signal output continuity and/or continuous first derivative properties. In the example, which swaps filters online, the new filter is operating out of sync with the original filter. A rapid switchover will create a discontinuity in both the signal and its first derivative.
2. The system can fail to maintain real-time constraints during reconfiguration. If the reconfiguration cannot be completed in sufficient time, deadlines will be sacrificed. In addition, the timebase can be shifted, resulting in a skew in system output period.

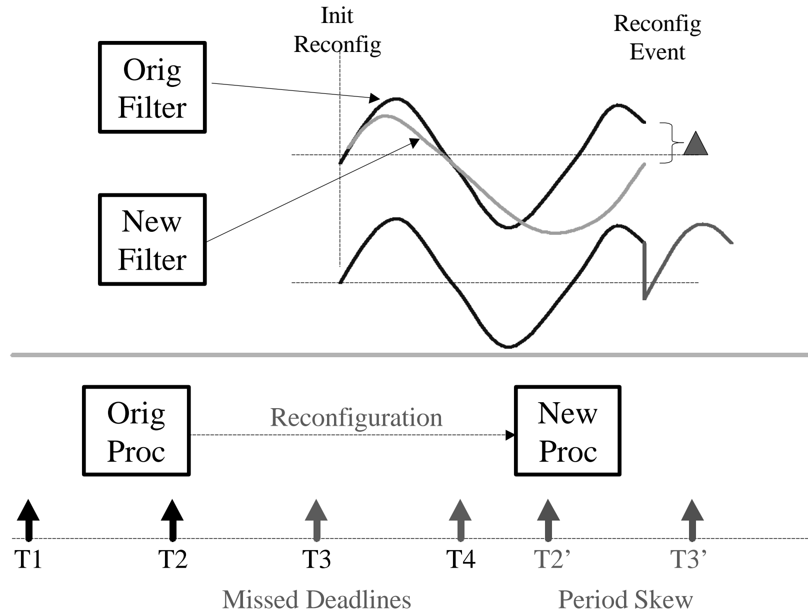


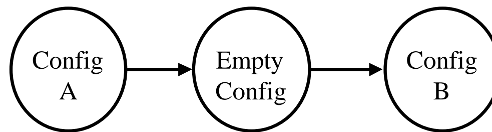
Figure RE3: Maintaining Application Consistency Through Reconfiguration

Runtime Reconfiguration Strategies

It is clear that reconfiguration support must be built into the design approach, from the lowest levels of the execution environment, to the high-level design/requirements capture tools. The extent of support is defined by the requirements of the target systems. The driving factors include how fast the system must reconfigure, whether intermediate states must be preserved (Application Signal Continuity), and if timing must be preserved. We now examine the potential reconfiguration strategies and their impact on system capabilities.

Reboot Strategy

The simplest reconfiguration strategy is termed the “Reboot” approach. It involves the orderly shutdown of tasks, bringing the system to a known, clean state. From this state, a new processing structure is constructed (Figure RE4). The implementation for this approach is simple, requiring the minimum amount of non-standard support from the execution environment and there is no need for additional processing capability for overlapping modes.



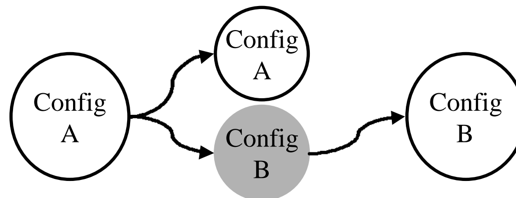
S(A) → S(NULL) → S(B)

Figure RE4: Reconfiguration Strategies – “Reboot” Approach

The drawbacks of this approach are severe. The system is offline during the reconfiguration time. No events can be handled, so a system under control is open-loop during that time. There is no provision for preservation of state. This can lead to long recovery times when the new configuration is started. Both of these factors lead to system application transients, both timing and signal continuity. This approach is not suited for the majority of embedded, closed-loop systems.

State Transition Approach

The second approach allows the insertion of transitory states between the major system operating modes (Figure RE5). These states allow the system to take smaller steps between operational modes to approximate a continuous-time transition, resulting in smaller transients. The intermediate configurations inherit state from their predecessors. The intermediate algorithms must be designed to gradually shift system behavior. While not continuous, the steps can be made arbitrarily small.



S(A) → S(A’) || S(B’) → S(B)

Figure RE5: Reconfiguration Strategies – State Transition Approach

This approach has several positive aspects. The state preservation allows transients to be minimized. The magnitude of the steps can be chosen by the designer to minimize key application behaviors. Few spare resources are needed, since the system is operating in only one mode at a time. The flexibility is limited only by the designers and by the time available for the transition.

There are several difficulties in this approach: The execution infrastructure must support the rapid transition of processes and transition of the states of the changing processes. The states must be mapped to the structures required by the next step, and installed with the new processing structure. The computation of the mapping may be complex.

The design of intermediate states can be complex, depending on the application. These transitory states depend both on the initial state and the final state, the algorithm characteristics, and the timing requirements. For smooth application transitions, many intermediate states may be required, leading to long transition times. (It should be noted that the application system is still under control during transition, but probably not the

optimal algorithm.)

Parallel State Transition Approach

An extension of the State Transition approach allows the system to execute several modes in parallel. This has the same benefits as the state transition approach with the added benefit of being able to execute algorithms prior to use, in an offline mode. The state of the offline process can be allowed to stabilize prior to impacting upon system performance. When transients have disappeared, the system can be transitioned to the new state (Figure RE6).

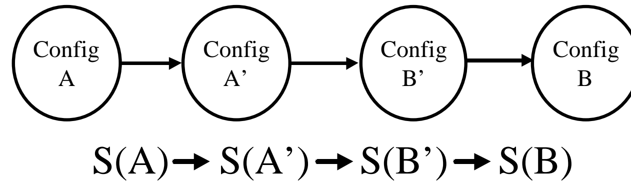


Figure RE6: Reconfiguration Strategies – “Parallel Execution” Approach

This approach has several benefits. The application-level transients can be minimized by proper design. The downtime is minimal, as is the operation of the system in a less-than-optimal configuration. Multiple states can be preserved, not forcing all information to be encoded in one format. This minimizes the impact of the design of one mode on another, thus simplifying design.

There are also several drawbacks. The underlying runtime environment must support mechanisms for rapid stepping between processes, the ability to execute multiple threads simultaneously, and the combination of attributes from the parallel executing processes.

System design is complicated by the need to design parallel structures. (In some cases, the parallel approach allows design separability, simplifying matters.) The necessary computational resources are increased, due to the need to execute multiple parallel processes.

Given the difficulties of implementation, the capabilities of this approach are required to service many reconfigurable application domains.

Execution Environment Design

The previous sections assembled a set of requirements for the execution environment. They also point out some of the design complexities. Working alone, the execution environment cannot solve these problems. The overall system design approach must span from the top-level algorithm designers/system requirement & resource specifications down to the hardware/software implementations. The top-level design issues have been discussed in terms of a domain-specific modeling environment, where the environment is tuned to reconfigurable system design. The Execution Environment forms the infrastructure onto which these designs are projected.

The Execution Environment must be designed with an interface suitable for synthesis from a MIC-Generator approach. The concepts, properties and interfaces of the runtime environment must be compatible with the design representation and synthesis approach. Capabilities and interfaces should be tuned to simplify the generator. This requirement demands a simple, uniform interface with a well-defined, consistent set of semantics that

apply throughout the system. Since the system includes software, hardware, and interactions between parallel modules, a common structure must map to a wide range of components.

The execution environment concepts have been driven by results from using tools developed over the past several years. These tools are currently 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 [4][14].

The semantics of the execution environment implement a large-grain-dataflow architecture. The Worker Function captures the tasks that are performed by the system. Communication nodes capture the transfer of data between workers. Computations can be described as a bipartite graph, where workers connect to Comm nodes, and Comm nodes connect to workers. At this level, there are no implied semantics of the workers. The execution properties of workers (Data tokens produced/consumed per execution, timing of execution, etc) are maintained at a higher level. The semantics of the Comm units are asynchronous queues.
When the generic large-grain dataflow graphs are implemented, they must be mapped down to a physical implementation. The implementation takes the form of either software or hardware. Software workers execute on a DSP or CPU, which we term Processes. Hardware workers are either implemented in reconfigurable hardware (FPGA's), ASIC implementations, or combinations of both. Processes and Processors are logically equivalent, representing functions on data. Processes/Processors are connected via logical Comm that must buffer, communicate, and match data formats. In software implementations, the Comm object is implemented by the OS/Kernel as a Stream, a software queue in memory. In hardware, the Comm object is implemented with registers and/or FIFO, or simply wires (Figure RE7).

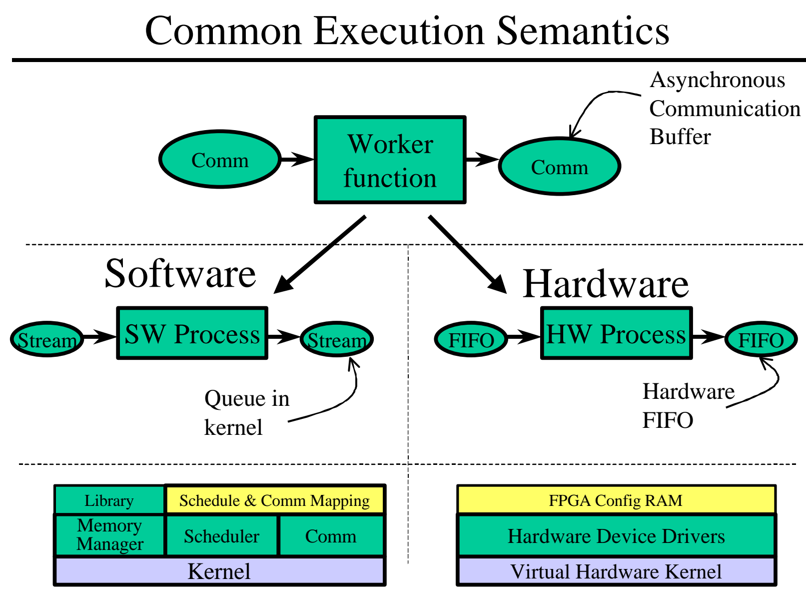


Figure RE7: Runtime Execution Environment: Common Execution Semantics

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 [14]. 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 kernel allows dynamic editing of the process table, and of the communications maps. The proper sequencing of these operations, including task execution phases, is necessary for the avoidance of reconfiguration problems. The current approach supports the "Reboot" approach directly, and will support the more advanced reconfiguration approaches with cooperation of the application tasks.
The hardware execution environment supports the same operational semantics. The implementation, however, is much different. The Virtual Hardware Kernel exists as a concept used in the system synthesis. The MIC Generator synthesizes a set of VHDL structural codes, one for each configurable device multiplied by the number of operational modes. Processors are directly synthesized using predefined components. Communications elements are selected from a library of interface types, based on the requirements of the workers on either end, the required performance, and the available resources. The communication infrastructure works in cooperation with the software communications, performing the signal buffering, and the necessary off-chip interfaces and data converters. The interface components are drawn from a library of modules. The modules implement a limited set of standardized communications protocols to transfer data between modules, and present data in the format required by the destination processor. As the system is used for more applications, the set of interface types will grow in capability.

Inherent in these interface components must be the capability to reconfigure. This involves strict synchronization mechanisms, methods for saving and restoring states, and facilities to allow function and structure modification. Global system synchronization is greatly aided by having a common system clock, and facilities for very low-latency signaling within the system. Our current concepts for reconfiguration require a single interrupt signal to be present at each component participating in a reconfiguration.

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.

Reconfiguration Manager

The reconfigurable hardware interfaces, and the flexible microkernel provide the facilities to implement system reconfiguration, however the problem of control and synchronization is critical. A global view of the system is necessary. Reconfiguration cannot be performed by the kernel alone.

This synchronization and control of a system during reconfiguration is the responsibility of the Configuration Manager. The CM contains tables capturing the behavioral state machine defined by the designers Behavioral Models. Tied to these state-based descriptions is the information necessary to configure the hardware and software components of the system.

Given this information, the Configuration Manager serves as a system observer. The CM monitors relevant signals, as defined in the transitions leading out of the current state. When the logical conditions for a state transition are satisfied, the Configuration Manager begins the structural transition process.

The first stage of the reconfiguration involves bring the system into a known, safe state. All communication interfaces must terminate. Since many of the data ports are bi-directional, the bus token must be returned to the 'safe' state. Computations must be completed and transitioned into the 'safe' state. The safe state may involve using local algorithms to perform the basic required functions to keep the system stable.

After all necessary components are in the safe state, the global interrupt is toggled to initiate the reconfiguration event. At this point, all communications must stop for the short period required for reloading the FPGA's bitfiles and the Software schedules and communication mappings. Since the state of the system was in a known safe state prior to reconfiguration enactment, there it little overhead atop the basic information download. The CM will reload the necessary FPGA's using the standard download methods. A sequence of commands is sent to each of the processors to enact the new processing graph and interface components. Once the new programming information is installed, the system interrupt signal is toggled to ensure a globally synchronized start up operation.

APPLICATION EXAMPLE

The design environment has been used for several applications. Here, we will describe an Automatic Target Recognition application for missiles.

The design process involves iteratively constructing the previously described categories of models that capture system design information. The ATR application design begins with a specification of requirements in the form of Behavioral Models. Figure AE1, AE2 show the top-level models for the missile behavior. From a start-up and system initialization phase (Figure AE1), the system waits in the Ready state for signals from the operator. The Seek Target signal will start the active system operation in a Lock-on Before Launch(LOBL) or Launch signal will cause the system to transition to a Lock-on After Launch(LOAL) mode. The system enters the Acquire Long-Range mode, in figure AE2, where a many-target acquisition is performed, and a target is selected. The system enters into the long range tracking, until either the track is lost, or proximity sensors signal the system transitions into a medium range mode. This process repeats itself for Mid-Range and Short-Range modes.

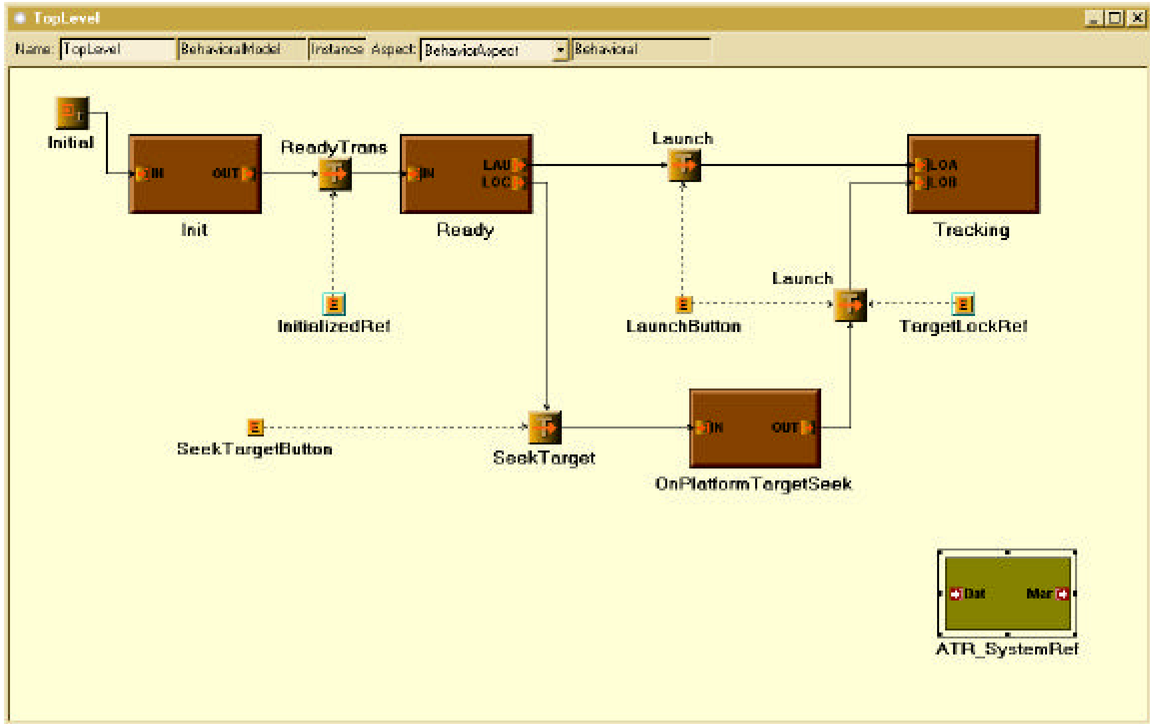


FIGURE AE1: TOP-LEVEL BEHAVIORAL MODEL

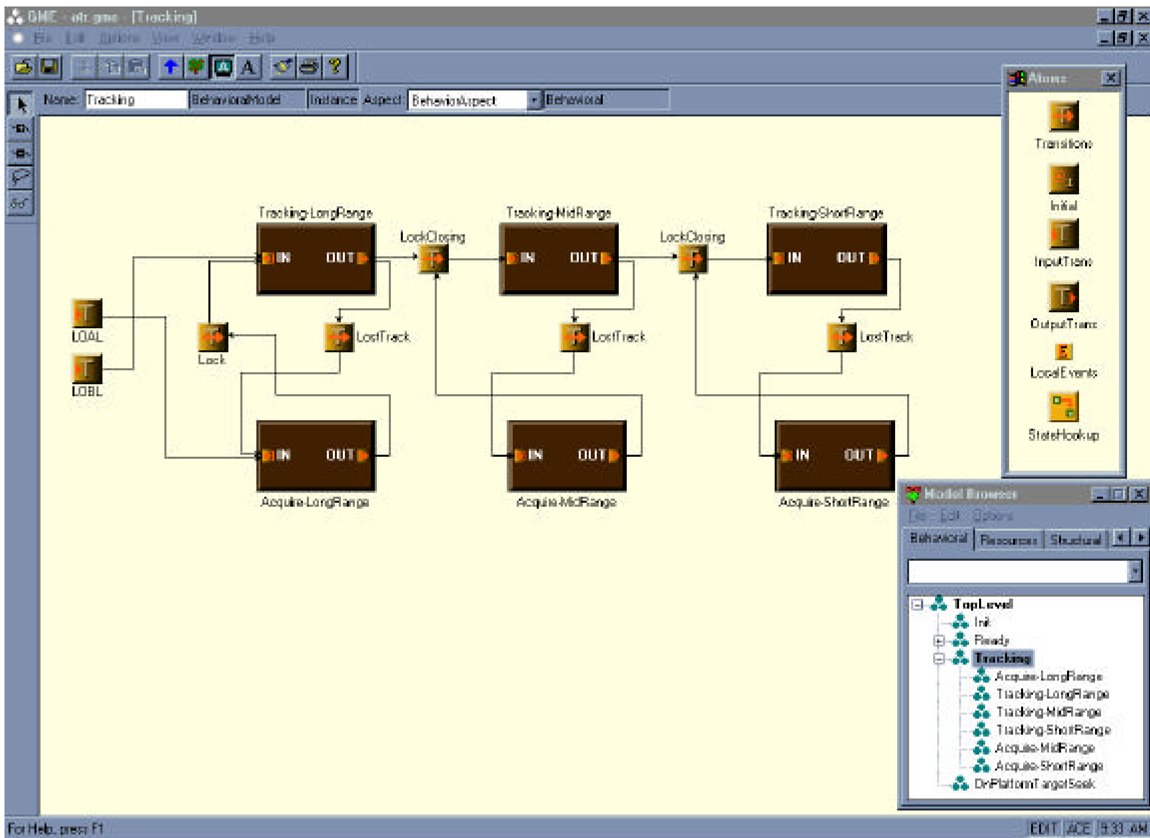


Figure AE2: ATR Behavioral Model, Tracking Drill-Down

Concurrently with the definition of the behavioral requirements, signal-processing engineers can define algorithm structures using a library of components. Hierarchy allows multiple designers to work at different levels in the design space. Figure AE3 shows the top-level signal flow for the long-range target acquisition modes. Figure AE4 shows the drill-down into a simple tracking algorithm for low-latency target tracking used in long range target tracking behavioral state.

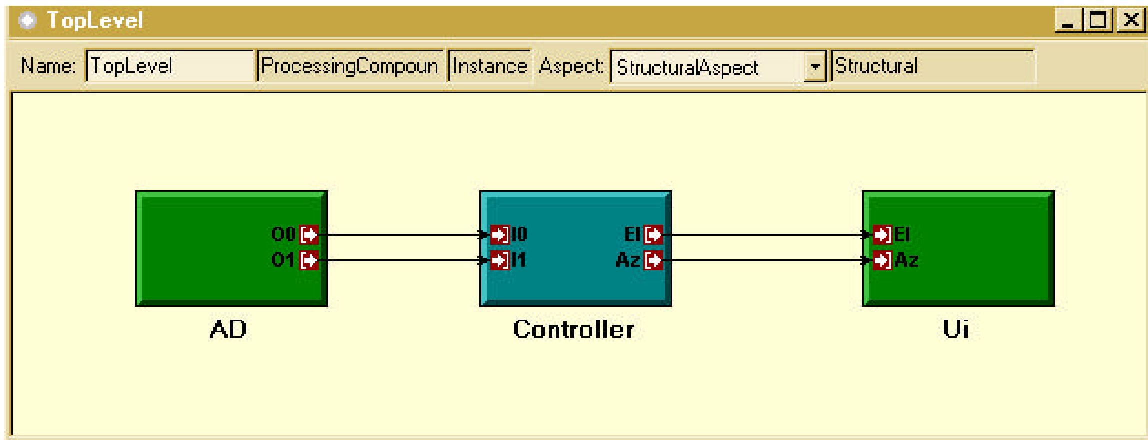


Figure AE3: Top-level Algorithm Structural Models

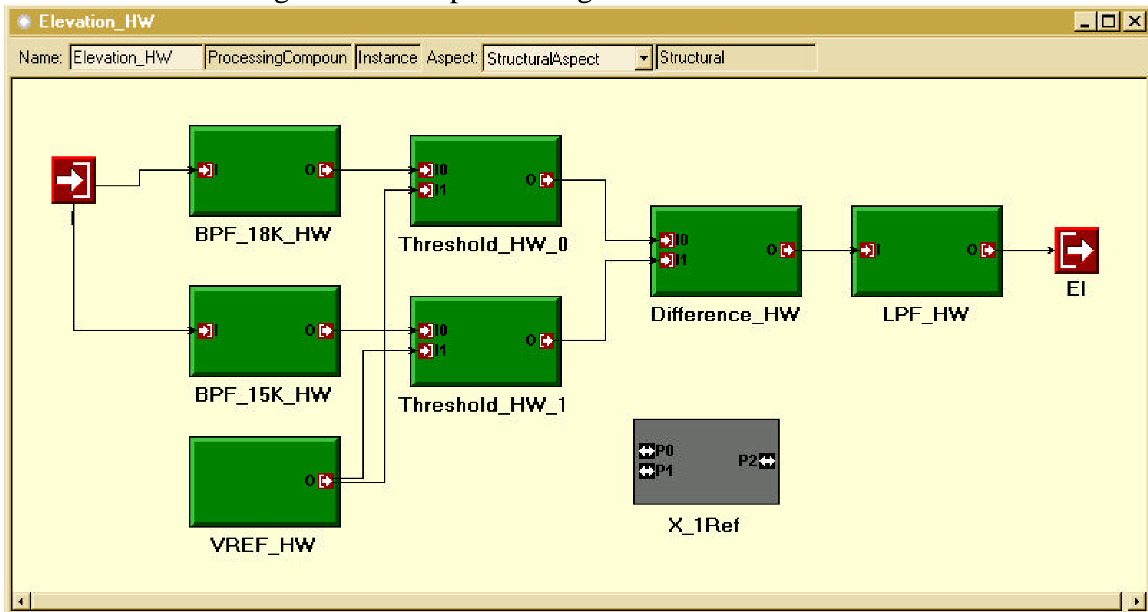


Figure AE4: Drill-Down into Tracking Algorithm Structural Models

These algorithms are described in a model hierarchy, using Compounds, Templates/Alternatives, and Primitives. Where possible, libraries of preexisting components are used. When new components are required, signal processing engineers and hardware VHDL designers develop or acquire modules and capture implementation attributes, such as benchmark results, into the models component libraries. In the tracking algorithm, several components were developed for hardware in VHDL and software (C

for the TMS320C40). The models show the IIR Bandpass filters, signal thresholding and differenceing, and low-pass filters.

Concurrently with the design of Behavioral and Algorithm Models, hardware engineers are capturing the hardware architecture details in the Resource Models. If the system is to be constructed with flexible hardware modules, the specifics of these modules are captured and the final assembly can be left for future specification. Where the boards are hardwired, the complete topology is captured directly. Figure AE5 shows the top level of the Resource models. This figure shows the 2 FPGA's, 2 DSP processors, 1 RISC processor and the A/D available for target tracking.

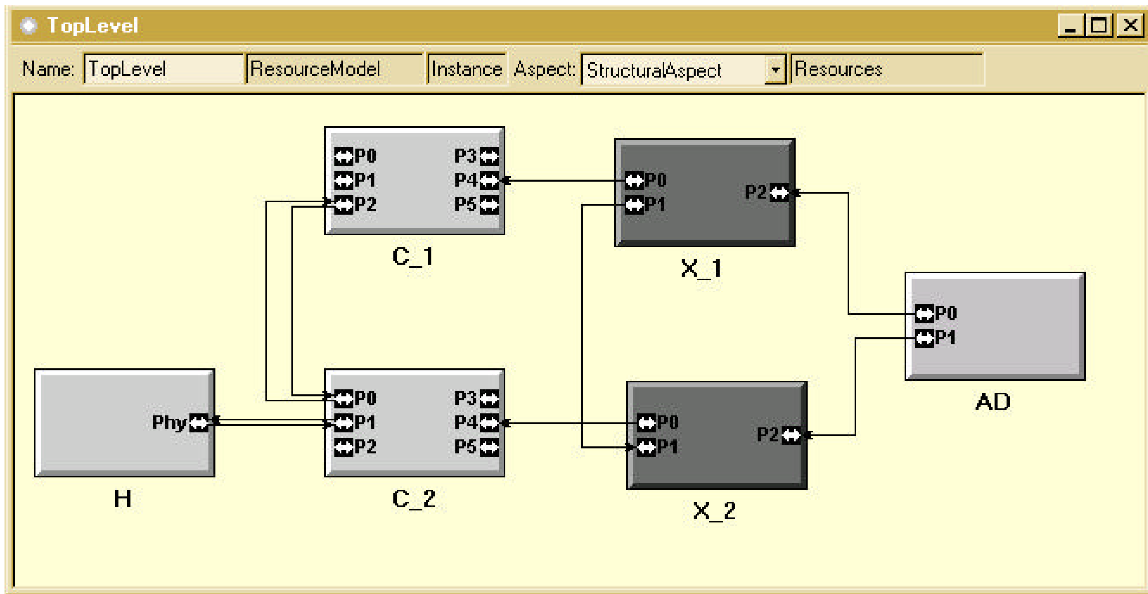


Figure AE5: ATR Hardware Resource Models

The component models are assembled by assigning Algorithm models to Behavioral Models, and assigning Resources to Behavioral Modes and Algorithms. Constraint specifications are developed to express complex relationships. See Figure AE6.

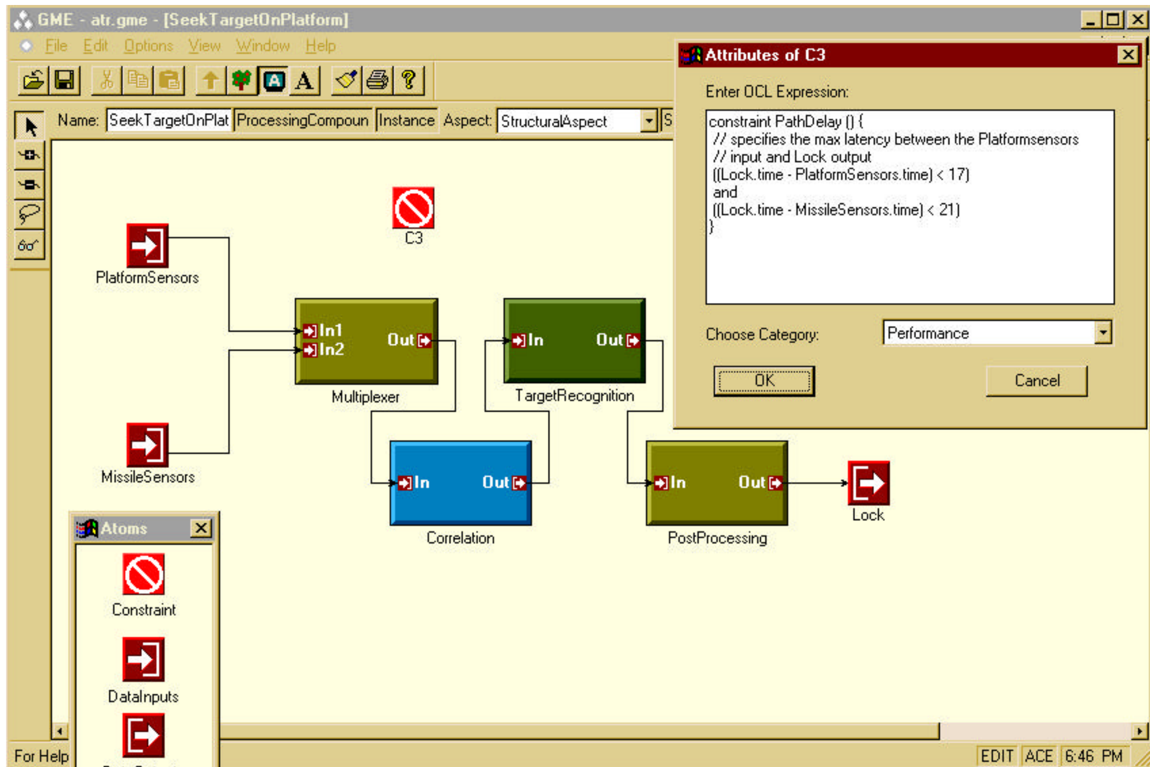


Figure AE6: ATR Constraint Specifications

The models are analyzed with the symbolic constraint manager to explore the design space. The initial design space in the ATR algorithm is 10^{24}. The constraints are iteratively applied to reduce the system to approximately 10 potential configurations.

From the remaining configurations, the designer selects one for implementation. The synthesis produces hardware architecture. The VHDL designs are compiled using Synopsis for Xilinx. The software structures are processed via the Texas Instruments C compiler.

Finally, the system is executed using the configuration manager's system loading tools. Figure AE7 shows a testbench configuration with internal signals displayed on a Windows-based user interface. Intermediate designs can be instrumented with graphical displays to view algorithm internal data structures.

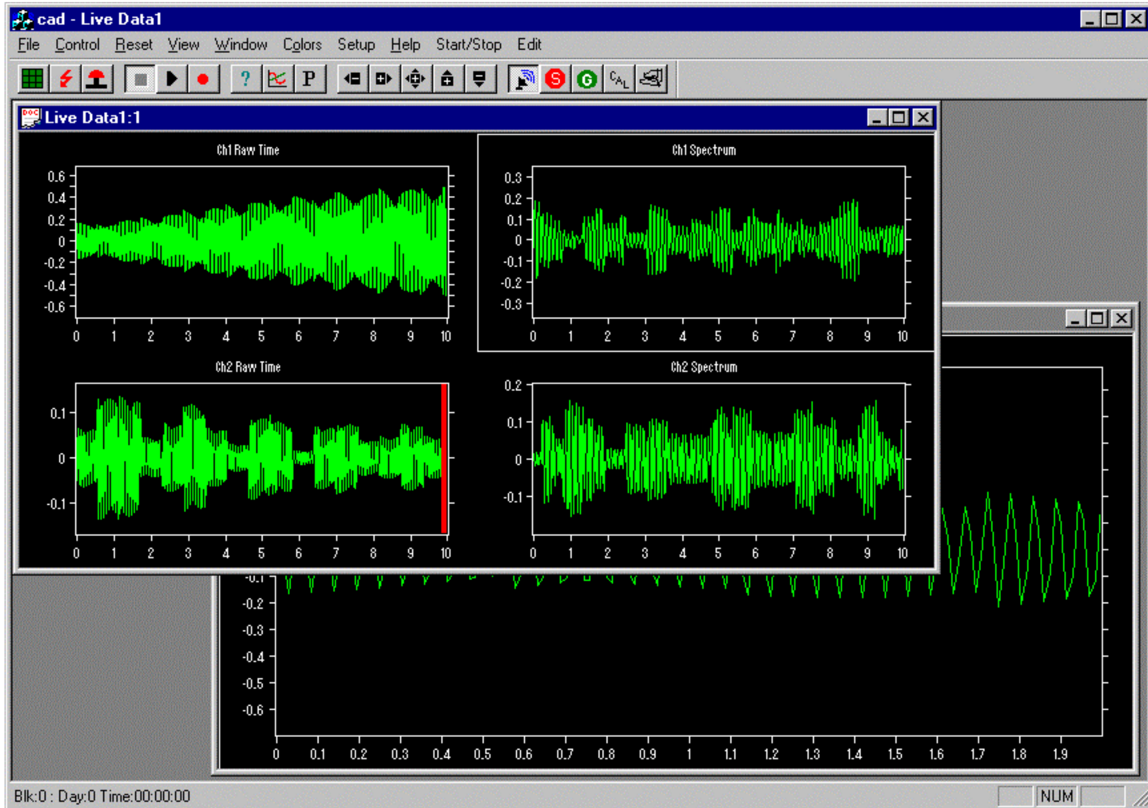


Figure AE7: ATR testbench display

This discussion shows one path through the design process. Typically, the process involves iterations, to optimize the algorithm performance, resource utilization, and system functional behavior.

CONCLUSIONS

The system described within this paper represents an ambitious set of goals for a design tool. The tool represents a comprehensive approach to the design of heterogeneous, real-time, resource-limited, dynamically adaptive systems. The Model-Integrated approach has been designed to support the many aspects and disciplines of embedded systems design. The flexible representation, analysis and synthesis of systems has the potential to reduce design effort and increase system flexibility. The underlying Runtime Environment, through the abstraction of hardware and software details, presents a uniform architecture for system implementation.

The prototype tool set has been applied to several small-to-medium-sized design projects with significant success. The tools are still research-quality and several key components are still in the process of design and implementation.

The design approach leads to flexible solutions. The implementation architecture is decoupled from the algorithm. Also, hardware is modeled as a set of generalized

resources. These two factors combine to support device technology evolution.

The high-level approach should produce greater design efficiencies. Given a rich set of component libraries, complex systems can assembled rapidly. The component libraries can be specialized to very high-level functions by the construction of hierarchical models. The availability of design alternatives within these functions will allow the efficiency of these components to be maintained near the level of a hand-coded system.

There are still many major research challenges to achieve a fully functional, robust design tool. These issues are:

1. Optimization: The current approach involves defining a very large design space and using constraint methods to extract a set of potential design solutions. The same type of evaluation concept is used in the simulation/evaluation approach. While these approaches can significantly reduce the design space (in the case of OBDD's) and can give several estimations of performance. For any one application, the process relies on the engineer to manipulate a complex, interrelated constraint networks. This process should be assisted further in the design environment. Simple tools are planned that show a sensitivity analysis of a user-defined performance function vs each of the user constraints. This will help to guide the designer to the appropriate constraints that impact system performance. Taking this a step further, optimization procedures can be implemented to automate the manipulation of system parameters and constraints. In such a non-linear, discretized space, no guarantee of optimization convergence is possible.
2. Methods for assessing the transient upsets that will occur during a structural reconfiguration are needed. These transients are needed for both numerical results and for the timing behavior.
3. Libraries and procedures for rapidly incorporating vendor IP must be available to ensure up-to-date components are available for the design. This also contributes to the ease of updating the technologies in the target platform.
4. Significant effort is required to transition the tools from a research prototype to a supportable, accepted design methodology and design environment.

REFERENCES

[1] Villasenor, J., Mangione—Smith, W., “Configurable Computing”, Scientific American, June, 1997.
[2] Arnold, J., Buell, D., Davis, E., “Splash 2”, Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992
[3] David R. Martinez, “Real-time Embedded Signal Processing”, IEEE Signal Processing Magazine, September 1998.
[4] Bapty, T., Ledeczi, A., Davis, J., Abbott, B., Hayes, T., Tibbals, T.: "Turbine Engine Diagnostics Using a Parallel Signal Processor", Joint Technology Showcase on Integrated Monitoring, Diagnostics, and Failure Prevention, Mobile, AL, 1996.
[5] Karsai G., Sztipanovits J., Padalkar S., DeCaria F.: "Model-embedded On-line Problem Solving Environment for Chemical Engineering", Proceedings of the International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, Florida, Nov. 6-10, 1995
[6] Long E., Misra A., Sztipanovits J.: "Saturn Site Production Flow (SSPF): Accomplishments and Challenges", Proceedings of the Engineering of Computer Based Systems, Maale Hachamisha, Israel, AL, March, 1998.
[7] Davis, J., Scott, J., Sztipanovits, J., Karsai, G., Martinez, M.: "Integrated Analysis Environment for High Impact Systems," Proceedings of the Engineering of Computer Based Systems, Jerusalem, Israel, April, 1998.
[8] Bapty T., Sztipanovits J.: "Model-Based Engineering of Large-Scale Real-Time Systems", Proceedings of the the Engineering of Computer Based Systems (ECBS) Conference, Montery, CA, March, 1997
[9] Carnes J. R., Misra A.: "Model-Integrated Toolset for Fault Detection, Isolation and Recovery (FDIR)", Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems, Friedrichshafen, Germany, AL, March 11-15, 1996
[10] Harel, D., “StateCharts: A visual Formalism for Complex Systems”, Science of Computer Programming 8, pp 231-278, 1987
[11] Bryant, R.E., “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams”, Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, June 1992
[12] Bryant, R.E., “Graph-based Algorithms for Boolean Function Manipulation”, IEEE Transactions on Computers, C35(8), 1986
[13] Kumar, S., F. Rose, "Integrated Simulation of Performance Models and Behavioral Models," Proceedings of the Fall 1996 VIUF, pp 185-194, Durham, NC, October, 1996
[14] Bapty T., Abbott B.: "Portable Kernel for High-Level Synthesis of Complex DSP-Systems", Proceedings of the the International Conference on Signal Processing Applications and Technology, Boston, MA, May, 1995
[15] Sandeep Neema: “Constraint based System Synthesis”, Technical Report, Department of Electrical and Computer Engineering, Vanderbilt University, 1999.

[16] Hein, C. and D. Nasoff, “VHDL-based Performance Modeling and Virtual Prototyping”, Proceedings of the 2nd Annual RASSP Conference, Arlington, VA, July 1995.
[17] James Rowson, “Hardware/Software cosimulation”, Proceedings of the 31st Design Automation Conference, pages 439-440, San Diego, CA, June 1994.
[18] Russel Klein, “Miami: A Hardware-Software cosimulation Environment”, Proceedings of the 7th IEEE International Workshop on Rapid Systems Prototyping, June 1996.