Survivability Using Adaptive Reconfigurable Systems

Azween Bin Abdullah†,

University Technology Petronas, Bandar Seri Iskandar, 31750 Tronoh, Perak, Malaysia

Summary
Significant advances have been made in reconfigurable computing device technology paving the path for fast, dynamically reconfigurable systems. The biggest challenge in the way of flexible adaptive computing is runtime reconfiguration. Many difficult issues are involved in runtime reconfiguration. Some of these issues are application specific and as such need to be considered by application designer while designing the system. Adequate high-level tool support is required to assist the designer in this complex task. Other difficult issues are concerned with the hardware and software consistency and the real-time behavior of the system. The execution environment must provide enabling support for these issues. The paper discusses the issues involved in runtime reconfiguration and presents a Model Integrated approach that attempts to address some of the issues at multiple levels. The Model Integrated approach provides a design environment that assists the application designer in capturing system design, requirements and constraints, and provides analysis and automatic synthesis capability from the captured information. The synthesized system is deployed in an execution environment that provides the basic primitives for reconfiguration and enables some of the difficult issues in runtime reconfiguration..

Key words:
Survivability, Reconfiguration, Runtime.

1. Introduction

Recent years have witnessed significant advances in programmable devices and technology. The Field Programmable Gate Array (FPGA) technology has come of age producing very high-density gate arrays (~1 million gates) with lower power consumption, faster clock-rate and relatively low configuration time [1]. A variety of hardware platforms comprising of general-purpose computers and FPGAs have been developed [2]. The presence of fast programmable circuitry has opened up a vast array of opportunities for application developers. A large number of high-performance computing applications have been successfully mapped to these platforms taking advantage of the fine-grain parallelism made available by the programmable hardware, and demonstrating significant speedup over the conventional general-purpose computer based implementation [3].

However, the promise of programmable device technology extends far beyond these point applications. The real potential of reconfigurable device technology lies in Adaptive Computing Systems (ACS) – systems that adapt and evolve in response to the changing environment while operational, without compromising the consistency and real-time properties of the system [4]. Several class of modern applications require very high performance with minimal resources and high flexibility to operate in a rapidly changing environment. Adaptive computing has been considered as an enabling approach for such applications. The essence of adaptive computing approach is to create several different configurations (hardware architecture/software topology) for a system, each tailored to a specific set of operational requirements. The system is reconfigured with a suitable configuration when the environment and thus the operational requirements change. The primary benefit of adaptive computing lies in being able to deliver high performance for a large number of operational goals with minimal hardware, by maximizing component utilization and minimizing hardware redundancy.

There are many challenges in the adaptive computing approach. The biggest yet the most understudied challenge however is runtime reconfiguration. Many difficult issues arise. Some of these issues are specific to the application being designed while others are more general and relate to the underlying runtime environment. While some studies in the adaptive computing community have demonstrated restricted runtime reconfiguration capabilities [5], most of the demonstrated approaches are highly application specific and lack adequate formalism. Additionally the primary focus of reconfiguration in these studies is programmable hardware. This clearly limits the approach's applicability, as most high-performance systems involve a mix of hardware and software components and heterogeneous technologies. Therefore a more comprehensive approach is required that considers the problem of runtime reconfiguration at multiple levels, starting from algorithms and all the way down to hardware details.

Our research attempts to address runtime reconfiguration in a more comprehensive and formal setting. We approach runtime reconfiguration at two levels: 1) Design – We have developed high-level design tools for representation, analysis and synthesis of dynamically reconfigurable systems. 2) Runtime – We have developed a uniform execution environment for execution

Manuscript received January 5, 2009
Manuscript revised January 20, 2009

of dynamically reconfigurable systems. In this paper we present runtime reconfiguration and our approach to runtime reconfiguration in a systematic manner. First, in section 2, we discuss the different scenarios and motivations for runtime reconfiguration and attempt a categorization of runtime reconfiguration based on these scenarios. Next, in section 3, we describe the difficult issues that arise, both at an application level and at the execution environment level. Then in section 4 we briefly describe our design tools and the uniform execution environment. Finally, in section 5, we conclude the report by evaluating our approach and recommending future directions.

2. Runtime Reconfiguration Categories

The challenges associated with runtime reconfiguration are closely linked with the goal of reconfiguration. Therefore, it is important to consider the motivation and the different scenarios of runtime reconfiguration. We have identified the objective of reconfiguration into three categories primarily. These categories are: a) Algorithmic reconfiguration; b) Architectural reconfiguration; c) Functional reconfiguration. They are briefly described below.

2.1 Algorithmic Reconfiguration

The goal in algorithmic reconfiguration is to reconfigure the system with a different computational algorithm that implements the same functionality, but with different performance, accuracy, power, or resource requirements. The need for such reconfiguration arises when either the dynamics of the environment changes or the operational requirements change. Figure 1 shows a scenario, where the dynamics of the controlled process change from 2nd order to 3rd order. In order to maintain the controllability the control equation employed by the controller also needs to change from 2nd order to 3rd order. Thus, while the functionality of the controller remains the same the control algorithm is different.

2.2 Architectural Reconfiguration

The goal in architectural reconfiguration is to modify the hardware topology and computation topology by reallocating resources to computations. The need for this type of reconfiguration arises in situations where some resources become unavailable either due to a fault, or due to reallocation to a higher priority job, or due to a shutdown in order to minimize the power usage. For the system to keep functioning in spite of the fault the hardware topology need to be modified and the computational tasks need to be reassigned. Figure 2 shows a scenario where architectural reconfiguration is employed. In this example an active FPGA device becomes inactive upon a mode transition. The computational tasks performed on that FPGA are relocated to another resource.

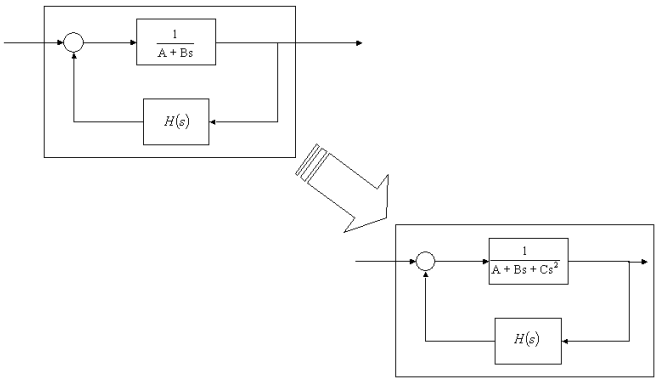

Fig. 1 Algorithmic reconfiguration of a controller by changing the control algorithm from 2nd order to 3rd order.

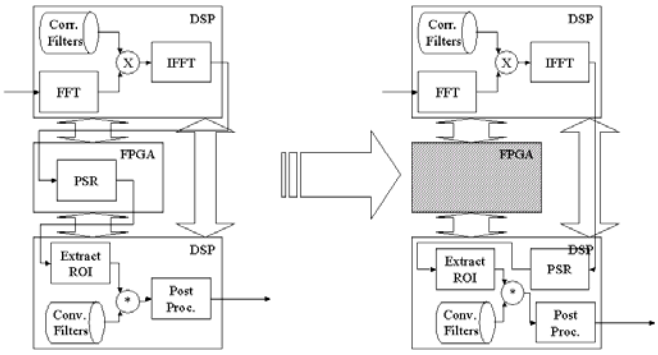

Fig. 2 Architectural reconfiguration of a computation by relocating computational tasks in the event of a resource failure.

This type of reconfiguration may also arise in an exactly opposite situation where new resources become available. In order to effective utilize the added resource the computational tasks must be reassigned and redistributed to resources.

2.3 Functional Reconfiguration

The goal in functional reconfiguration is to execute different function on the same resources. The need for this type of reconfiguration arises in situations where a large number of different functions are to be performed on a very limited resource envelope. In such situations the resources must be time-shared across different computational tasks to maximize resource utilization and minimize redundancy. Figure 3 depicts a scenario where functional reconfiguration is employed. In this example the FPGA device implements an ALU function in one mode of operation. The device is reconfigured to

implement a limiter function in another mode of operation. The reconfiguration in this scenario helps improve the functional density of the FPGA device. The Dynamic Instruction Set Computer at BYU [6] employs this type of reconfiguration.

Typical adaptive systems would be employing one or more of these reconfiguration categories to implement a dynamically adaptive behavior.

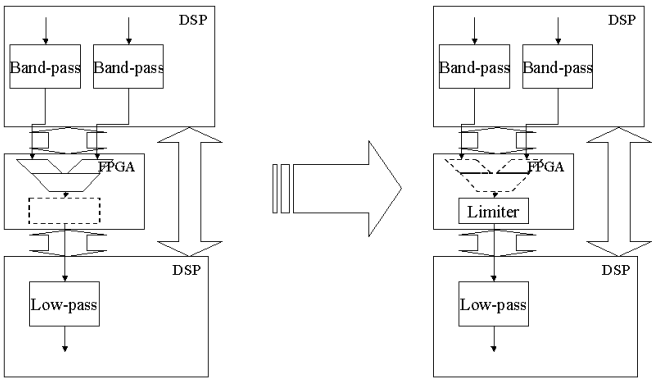

Fig. 3 Functional Reconfiguration of FPGA device to improve the functional density.

3. Challenges in Runtime Reconfiguration

The purpose of the categorization presented above was to distinguish different reconfiguration scenarios that may occur in real systems. These different reconfiguration scenarios present different challenges. We present these issues categorized into application specific and runtime environment specific below:

3.1 Application Specific Issues

Some of the challenges are highly specific to the application being designed and as such need to be considered and addressed by the application designer. The runtime environment must provide support API to enable these issues:

1) Issues in Algorithmic Reconfiguration

The first and foremost consideration, when dynamically adapting the algorithm is the continuity and transients in the output signal. Due to reconfiguration there might be a discontinuity in the output signal. An application designer needs to consider the magnitude of the discontinuity, the magnitude of the transient, the duration of the transient, and their impact on the overall system. The magnitude of the discontinuity and the reconfiguration transients depend on the nature of the algorithm. With filters it has been observed that the resonator structure has low reconfiguration transients [7]. With some applications it is possible that the discontinuity is entirely unacceptable, in which case alternative approaches need to be considered to maintain the continuity in spite of reconfiguration. Figure 4 shows a scenario where the output continuity is lost due to reconfiguration

The second consideration in algorithmic reconfiguration is the state mapping. There could be infinite state variable descriptions representing the same transfer function. For reconfiguration the internal state variable description of one algorithm needs to be mapped to the internal state variable description of another algorithm. This is a complex problem and needs to be understood by the application designer.

There are some other difficult issues that are related to the runtime environment and are presented later.

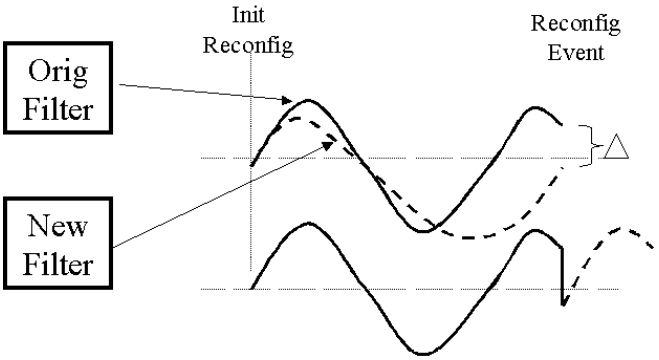

Fig. 4 Output discontinuity in algorithmic reconfiguration.

2) Issues in Architectural Reconfiguration

Architectural reconfiguration involves relocating computations. The important consideration here is the safe transitioning of the internal state of the computation. The state transitioning could be further complicated due to relocation of computation from hardware to software. The application designer needs to understand the trade-offs in the accuracy, and the performance of the algorithm. Additional difficult issues are involved in shutting down a computation on a resource and relocating it to another resource that need to be addressed by the runtime environment.

3) Issues in Functional Reconfiguration

Functional reconfiguration involves time-sharing of resources between computational tasks. The primary consideration here is to preserve the state and the intermediate results when a computation is swapped out. The internal state and the intermediate results must be restored when the computation is swapped back in. To address these issues the application designer needs to consider the internal state and intermediate results

representation and the runtime execution environment needs to provide API for preservation and restoration of state and data.

3.2 Runtime Environment Specific Issues

In addition to the application specific issues presented above there are some issues that are common to all types of reconfiguration and need to be addressed by the runtime environment. These issues are presented below

1) Synchronization

In order to partially or fully reconfigure an executing system all the concurrent computations executing on different resources need to be synchronized and brought in a consistent state. All the communications in transit must be finished. For the partial reconfiguration of a heterogeneous resource network, the segment of the network under reconfiguration must be shutdown or decoupled from the rest of the network. The synchronization task becomes even harder in the absence of a separate synchronization event bus, as the internal communication paths of the network that is about to be reconfigured are required to propagate the reconfiguration event.

2) Hardware Consistency

A host of problems can occur during reconfiguration that may destroy the consistency of the underlying hardware. Loss of hardware consistency can have many negative effects, ranging from temporary loss of performance to hardware damage and total/permanent system malfunction. Some of the possible scenarios are: 1) Port contention may occur when bi-directional ports are improperly initialized, a reconfiguration event is not properly sequenced / synchronized or if an improper/inconsistent design is implemented. If two connected drivers are enabled, permanent physical damage can occur to the circuits. 2) Data token loss or duplication can result from incorrect initialization or a loss of communication integrity, where 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. 3) Reconfigurable devices must maintain state when controlling a complex external hardware device, such as an attached processor or storage device. If a reconfiguration occurs during a state transition within the device, or if the reconfiguration incorrectly 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. Figure 5 depicts these scenarios.

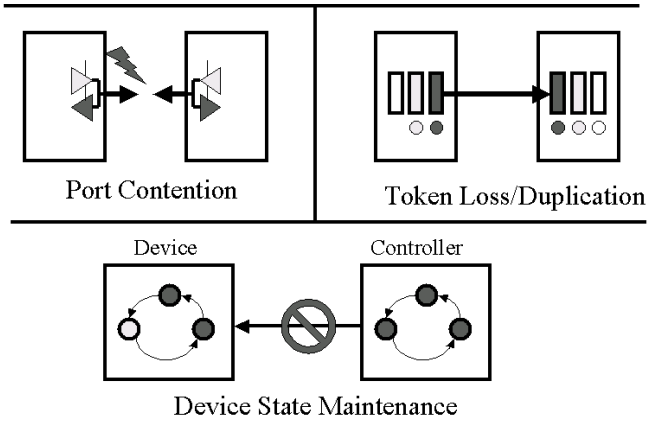

Fig. 5 Possible scenarios of loss of hardware consistency on reconfiguration.

3) Software Consistency

Software issues can present a larger challenge to dynamic system reconfiguration. The internal state of software must be managed under reconfiguration. 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. As a result memory leaks could occur that would adversely affect long-term reliability. Task structure mismanagement can happen resulting in extra tasks executed by the kernel, with a loss in performance. In a message passing system messages in transit can be delivered when the receiving process no longer exists, resulting in mismatched messages and communication errors. Figure 6 depicts a scenario of loss of software consistency on reconfiguration

4) Timing Consistency

The timing constraints of the application must be obeyed during reconfiguration. During reconfiguration the system can fail to maintain real-time constraints. If the reconfiguration cannot be completed in sufficient time, deadlines will be sacrificed. In addition, the time-base can be shifted, resulting in a skew in system output period. Figure 7 depicts a scenario of loss of deadline resulting in timeline skew.

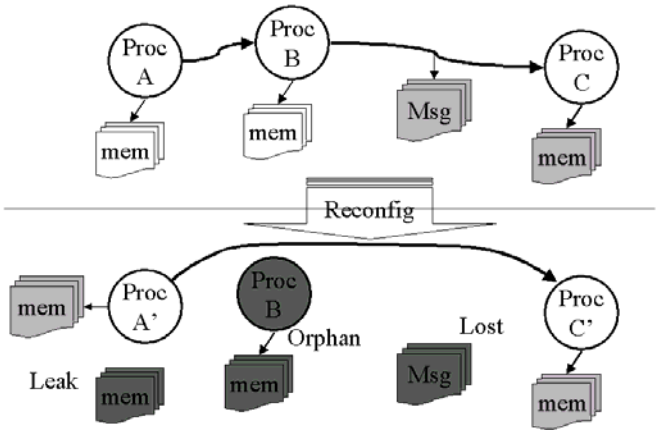

Fig. 6 Possible scenarios of loss of software consistency on reconfiguration.

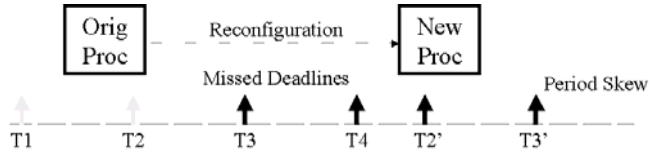

Fig. 7 A scenario of loss of deadline resulting in period skew on reconfiguration.

4. Model Integrated Approach

From these issues it is evident that runtime reconfiguration needs to be addressed at multiple levels. There should be a formal design environment with support for analysis and synthesis. Along with the design environment there should be a runtime environment that can be targeted for synthesis from design environment and that supports runtime reconfiguration. The Adaptive Computing Systems project at Vanderbilt University attempts to develop such a coupled design and execution environment. The design and execution environment are briefly described below.

4.1 Design Environment

The design environment is based on Model Integrated Computing (MIC), an approach for synthesizing computer based systems, developed and matured over a decade [8]. In MIC domain specific modeling environments are developed, those allow a designer to "model" the system in the concepts and formalism of the particular engineering domain. The modeling environment is a multiple aspect graphical editor that directly supports domain specific modeling concepts. Multiple view models capture the information relevant to the system being designed. The system models can be analyzed and verified by different static and dynamic analysis tools. The integrated models are also used for synthesis of runtime system. Figure 8 depicts the overall design flow in the MIC approach.

The main elements of the design environment are: 1) Design representation; 2) Design analysis and verification; and 3) Design synthesis. These are briefly described below.

1) Design Representation

The environment provides modeling objects (models, atoms, references, and connections) to capture different aspects of a reconfigurable system in familiar, well-understood models of computation. They are:

a) Behavior modeling: The reconfigurable system operates in discrete modes, with specific transitions between modes. Therefore, a Statechart representation is chosen to model the adaptive operational behavior [9]. The modeling objects provided are states, events and transitions. States represent operational modes, events represent the cause of a mode-shift, and transitions and transition rules define the pre-conditions and the consequences of a mode-change. States can contain states, events, and transitions thus enabling creation of a hierarchical finite state machine. The computation to be performed in a mode (state) is represented by referencing a computation model (described below). A reference is essentially a "pointer" to another model. Referencing allows a single computation to be applied to any number of system modes. States are annotated with attributes such as real-time specifications on the computation, power restrictions, and other user-defined constraints.

b) Computation modeling: The computations are modeled as a Dataflow Diagram, a well-familiar representation particularly suited to signal processing applications [10]. The modeling environment extends the basic Dataflow Diagrams with the concept of hierarchy and alternatives to add flexibility to the dataflow representation and help manage system complexity. Dataflow structures are modeled with the following modeling objects: primitives, compounds, and templates. A primitive object represents elemental computation. It maps directly to a hardware macro or a software function. Primitive objects are annotated with attributes that capture measured performance, resource requirements (memory/processing time/logic blocks), and other user-defined properties. A compound is an aggregation object that may contain primitives, other compounds, and/or templates. These objects are connected within the compound to define the dataflow. A template is a modeling object that models a processing block with rigorously defined interface and one or more alternative algorithm/implementation. These design alternatives can be compounds, primitives, or templates, allowing

hierarchies of design alternatives. With alternatives one can compose a very flexible design space, with a huge number of potential design implementations.

c) Resource modeling: The resource network is modeled as an interconnected network of processing elements. The modeling objects provided include processors, FPGAs, ASICs, datasources, and datasinks. Processors, FPGAs, and ASICs represent the so-named real-world objects. Datasources and datasinks capture the specifics of data acquisition/effector interfaces. These objects are annotated with the inherent performance attributes of the processing element such as clock speed, memory, logic blocks and other resources. The physical connection points on a chip are modeled as ports of these objects. Ports are annotated with communication protocols and pin assignments. Physical connections between processing elements are represented by connections between ports.

d) Constraint modeling: The modeling environment provides a constraint modeling object, that is annotated with a constraint expression specified in a language similar to Object Constraint Language (OCL). The constraint language provides a rich expressive medium by which the designer can express system requirements (latency, throughput, power etc), design rules (assign a computation to a particular resource) as well as guidelines for alternative selection and design implementation. These guidelines can be very valuable aid in the design space analysis and pruning described below. Figure 9 shows a screen capture of an example system modeled in the environment.

2) Design 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 design tools is to assist the designer in selecting 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 1024th).

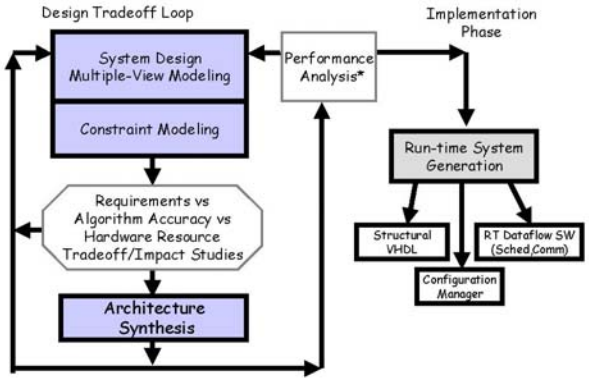

Fig. 8 Design Flow in the ACS Model Integrated Design Environment.

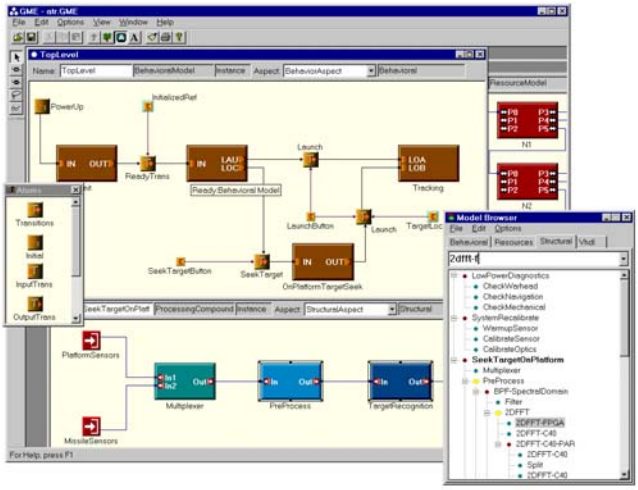

Fig. 9 Multi-aspect graphical Models.

To deal with such large spaces we have incorporated a multi-stage multi-resolution design analysis in the environment that starts with a large number of design alternatives and progressively refines the design space to a small set of best choices. As the number of designs under evaluation is reduced, the resolution of the analysis/estimation increases to get more accurate performance information.

The first stage of the analysis is an Ordered Binary Decision Diagrams (OBDD) [11] based analytical tool that symbolically evaluates the design space against the user-defined constraints. The principal benefit of a symbolic evaluation is that it does not require enumeration of the design space, which could be prohibitively expensive given the size of the space. The constraints can be selectively applied in this tool to eliminate the designs that fail to meet the system requirements, thereby pruning

the design space.

While symbolic constraint evaluation excels at large design space exploration, it lacks the ability to assess the fine performance details of system design tradeoffs. For that reason, the next stage of the analysis process is a multi-resolution performance simulation facility. The performance simulation facility is constructed using the Performance Modeling Library of Honeywell Advanced Research Division [12]. The system models are translated into a PML specification, which is then simulated in a discrete event VHDL simulator. The results of the simulation are translated back to the modeling environment for use in determining if the design satisfies performance specifications.

3) Design Synthesis

The result of design analysis process is a set of prescribed system modes, hardware architectures, and software structures (one set of architectures/software per mode). These results are represented in specific combination of the previously described models. These models must be processed to create the actual executable hardware/software product. A model interpretation process performs this task. The process is briefly described below.

a) Configuration manager synthesis: The adaptive behavior described in the behavior models is translated to a C-based state machine representation. The states in this state machine contain links to the configuration files (hardware/software). The configuration manager executes transitions in the state machine by reconfiguring the system with the appropriate configurations.

b) Hardware synthesis: For the configurable devices in the network, VHDL descriptions are generated. The VHDL design incorporates computational components from the design library glued together using components from a standard interface runtime library. The VHDL specifications are compiled using vendor-supplied/COTS compilers and device specific Place-and-Route tools to configuration specifications for the device.

c) Software Synthesis: For the general-purpose processors in the network, software architecture specifications are generated. These specifications provide the information needed by the low-level operating system to enact the desired computational behavior. Specifications include software load modules, real-time schedules, communication maps, and interfaces between software and hardware modules. 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.

4.2 Runtime Environment

The Runtime Environment is designed such that it can be easily synthesized from the high-level model-integrated design environment. The concepts, properties and interfaces of the runtime environment are made compatible with the design representation and synthesis approach. Capabilities and interfaces are tuned to simplify the generator.

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 (Figure 10). 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.

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 [13]. 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 very different. The Virtual Hardware Kernel exists as a concept used in the system synthesis process. The model-integrated design environment 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. 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.

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 that defines the transitions at which reconfiguration is to occur. 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.

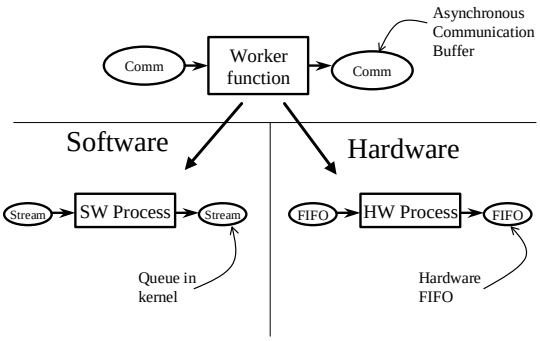

Fig. 10 Common Execution Semantics.

The first stage of the reconfiguration involves bringing 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 configuration files and the kernel's software schedules and communication mappings. Since the state of the system was in a known safe state prior to reconfiguration enactment, there is little overhead atop the basic information download. The configuration manager will reload the necessary FPGA's using the standard download methods. To enable the new processing graph, a sequence of commands is sent to each of the processing elements and interface components. Once the new programming information is installed, the system interrupt signal is toggled to ensure a globally synchronized start up operation.

3. Conclusions

The real promise of programmable device technology lies in flexible Adaptive Computing Systems. Adaptive computing promises high performance for a large number of operational goals with minimal hardware, by maximizing component utilization and minimizing hardware redundancy. However runtime reconfiguration poses a major challenge in the implementation of dynamically adaptive systems. Difficult challenges arise both at an application level and at the execution environment level. These challenges can be met only by considering runtime reconfiguration problem at multiple levels starting from design all the way down to implementation in a single thread.

Model Integrated Computing presents a unified approach to runtime reconfiguration. The design environment captures the design, requirements and constraints in models. Analysis and synthesis tools in the environment generate executable systems from the information represented in models. The flexible representation, analysis and synthesis capabilities of the environment have the potential to reduce design effort and increase system efficiency.

The runtime environment described provides a reconfigurable execution platform. Currently a simple reconfiguration strategy is supported that involves shutting down the executing computations, bringing the entire system into a consistent state and then starting the next set of computations. More complex reconfiguration strategies are planned. While the runtime execution platform described provides stable, predictable results and provides application consistency (e.g. no data loss) throughout reconfiguration, many more of the application specific issues described above need to be addressed.

References

[1] "Virtex™ 2.5 V Field Programmable Gate Arrays", http://www.xilinx.com/virtex.

[2] Waugh T. C., "Field programmable gate array key to reconfigurable array outperforming supercomputers", Proceedings of the IEEE 1991 Custom Integrated Circuits Conference, pp 6.6/1-4, 1991.

[3] Graham P. and Nelson B., "Genetic Algorithms In Software and In Hardware – A Performance Analysis Of Workstation and Custom Computing Machine Implementations", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, pp 216-225, April 1995.

[4] Howard N. and Taylor R. W., "Reconfigurable logic: technology and applications", Computing Control Engineering Journal, pp 235-240, September 1992.

[5] Eldredge J. G. and Hutchings B. L., "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, pp 180-188, April 1994.

[6] Wirthlin M. J. and Hutchings B. L., "A Dynamic Instruction Set Computer", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, pp 99-107, April 1995.

[7] Kovacshazy T. and Peceli G., "Transients in adaptive and reconfigurable measuring channels", Proceedings of International Symposium on Measurement Technology and Intelligent Instruments, Miskolc, Hungary, September 1998.

[8] Franke H., Sztipanovits J., Karsai G.: "Model-Integrated Computing", Proceedings of the 1997 Hawaii Systems Sciences Conference, (no page number available, CD-ROM publication), 1997.

[9] Harel, D., "StateCharts: A visual Formalism for Complex Systems", Science of Computer Programming 8, pp 231-278, 1987.

[10] Hatley D. J. and Pirbhai I. A., "Strategies for Real-Time System Specification", Dosret House, 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] Hein, C. and D. Nasoff, "VHDL-based Performance Modeling and Virtual Prototyping", Proceedings of the 2nd Annual RASSP Conference, Arlington, VA, July 1995.

[13] Bapty, T., Abbott, B., "Portable Kernel for High-Level Synthesis of Complex DSP-Systems," *Proceedings of ICSPAT '95*, Boston, MA, October, 1995.

Azween Abdullah obtained his bachelors degree in Computer Science in 1985, Master in Software Engineering in 1999 and his Ph.d in computer science in 2003. His work experiences includes eighteen years as a lecturer/senior lecturer in institutions of higher learning and as director of research and academic affairs at two institutions of higher learning, twelve years in commercial companies as Software Engineer, Systems Analyst and as a computer software developer and IT/MIS and educational consultancy and training. He has many years of experience in the application of IT in business, engineering and research and has personally designed and developed a variety of computer software systems including business accounting systems, software for investment analysis, website traffic analysis, determination of hydrodynamic interaction of ships and computation of environmental loads on offshore structures. His area of research specialization includes system survivability, formal specifications and modeling and software engineering.