RASSP Model Year Architecture (RASSP-MYA) Appnote

Abstract

The Rapid Prototyping of Application-Specific Signal Processors (RASSP) program is changing the way engineers design embedded signal processors. The objective of the program was to reduce time-to-market and cost by at least a factor of four, and improve design quality by at least a factor of four. These improvements were achieved by developing complementary methodology, model-year architecture and enterprise system infrastructure elements which, when combined, significantly reduce the cost and time required to field embedded signal processors. The model-year architecture defines how elements must be designed to achieve low-cost upgrades through the use of open interface standards. LM/ATL has defined and demonstrated a Standard Virtual Interface (SVI) and a Reconfigurable Network Interface (RNI) that when used properly can greatly reduce your product upgrade cycle time and cost

Purpose

The ATL RASSP team developed the model-year architecture to promote design upgrades and reuse via standardized, open interfaces while leveraging state-of-the-art commercial technology developments. What drives the RASSP signal processor architecture are the requirements imposed on signal processors to meet changing mission-critical processing needs, and the military's requirements for long-term life cycle support.

Roadmap

RASSP Model Year Architecture (MYA) Appnote

1.0 Executive Summary

The ATL RASSP team developed the model-year architecture to promote design upgrades and reuse via standardized, open interfaces while leveraging state-of-the-art commercial technology developments. What drives the RASSP signal processor architecture are the requirements imposed on signal processors to meet changing mission-critical processing needs, and the military's requirements for long-term life cycle support. RASSP must also address the full spectrum of signal processing applications, both commercial and military:

- Low-cost commercial applications, such as cellular communications and HDTV, which use anywhere from 1 - 10 processors
- Very large military sensor systems, such as shipboard radar systems, which use anywhere from 100 - 1000 processors.

This range of requirements imposes a formidable challenge: defining an architectural approach that addresses low-cost technology insertion, upgradability, and extensibility.

Model-year architectures must support scalability, heterogeneity, open interfaces, modular software, life cycle support, testability, and system retrofit. The notion of model-year upgrades is embodied in reuse libraries and the methodology for their use. The hardware and software elements within the libraries are "encapsulated" by functional wrappers. Adding this level of abstraction hides implementation details. It also makes practical technology insertion, reuse, trade-offs, and optimizations for specific applications.

Recently, the applicability of this technology was extended into the System-on-a-Chip arena. This technology is the basis for the Virtual Socket Interface Alliance [VSIA] bus wrapper technology, enabling system designers to quickly interconnect Intellectual Property (IP cores) from different vendors on a single chip.

RASSP Model Year Architecture (MYA) Appnote

2.0 Introduction

This application note is written for designers and program managers interested in defining an open architecture that supports low cost upgrades through technology independent *functional* interfaces. After reading this application note, designers will know how to:

- decide where and what type of MYA interface to insert in a design by analyzing candidate architectures to define proper subsystem-level interfaces that benefit from regular upgrades
- access the specifications which define the functional interfaces
- access the templates and guidelines that aid in implementing physical instantiations of the hardware and software using the MYA concepts and corresponding RASSP tools
- determine a software programming approach and evaluate the MYA software architecture for applicability
- apply appropriate interfaces to instantiated architectures, and identify RASSP tools to support hardware and software developments

Program managers will see how this approach will reduce the cost of technology upgrades by decreasing design and verification time through the use of standard, verified, interfaces. They will also understand the required investment in upfront planning and tools in order to apply this methodology effectively.

The model-year concept is analogous to the practices of the auto industry, where products are upgraded and improved on a model-year basis; complete products are redesigned less frequently. The key to reducing time-to-market and cost is to leverage commercial technology that provides significant processing improvements every 2 - 3 years. Embedded signal processors must support introducing these emerging processor components, products, and standards throughout the product life cycle, with minimal impact on the surrounding system hardware and software. These improvements are enabled by capturing and validating the processor's functional and performance characteristics at the system/subsystem level, independent of the specific hardware and software implementation. The goal of the model year architecture task is to provide a framework for developing application - specific signal processor architecture designs that encourages and facilitates hardware and software reuse, upgradability and technology insertion. This approach to plug-and-play technology is being adapted for use by the Virtual Socket Interface Alliance [VSIA] for interconnecting Virtual Components from different IP vendors in System-on-a-Chip applications.

The model-year architecture enables engineers to design architectures through a structured framework that ensures that designs incorporate all the required model - year features. The basic elements that comprise the model-year architecture are the functional architecture (hardware and software), encapsulated library components, and design guidelines and constraints, as shown in Figure 2 - 1. Synergism between the model-year architecture framework and the RASSP methodology is required because all areas of the methodology, including architecture development, hardware/software codesign, reuse library management, hardware synthesis, target software generation, and design for test, are impacted by the model-year architecture framework.

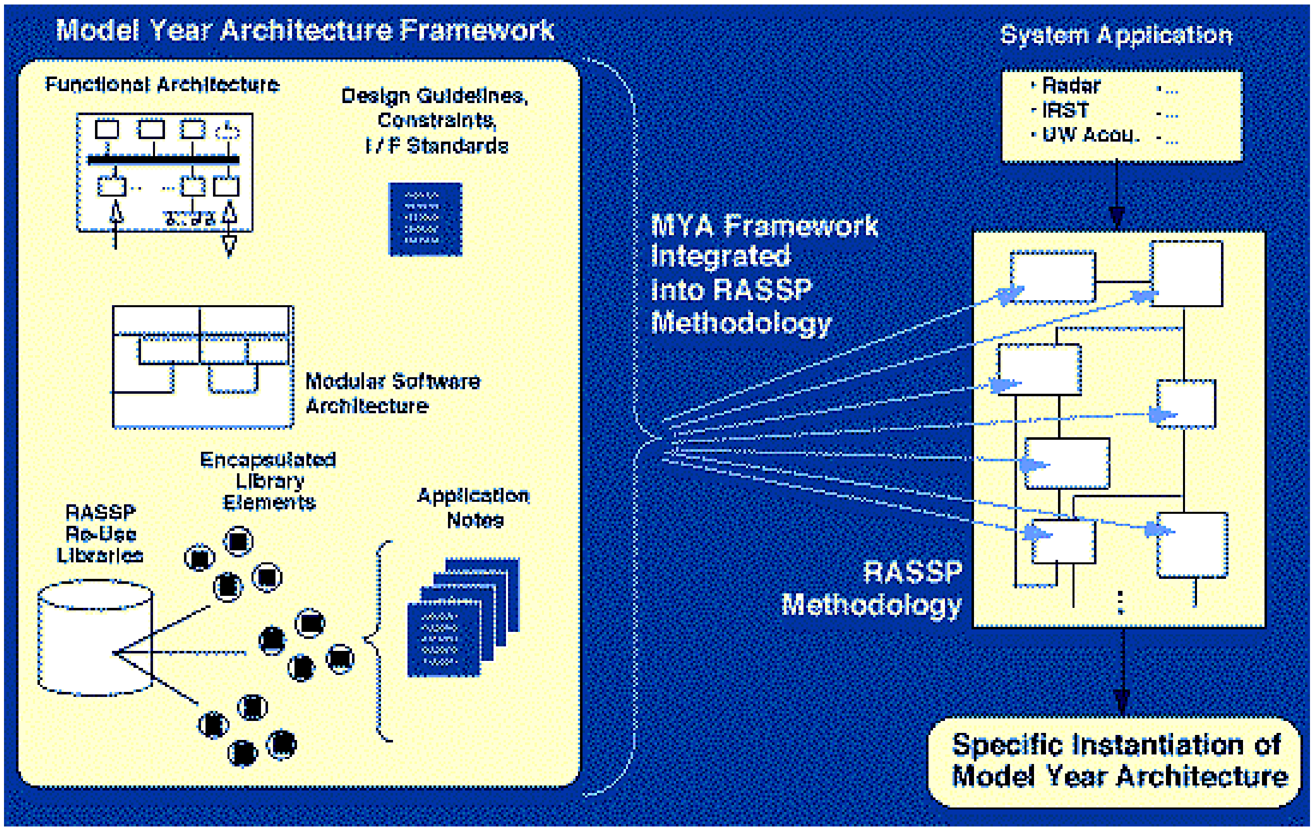
Model Year Architecture Framework
Functional Architecture
Design Guidelines, Constraints, I/F Standards
Modular Software Architecture
Encapsulated Library Elements
RASSP Re-Use Libraries
Application Notes
MYA Framework Integrated Into RASSP Methodology
System Application
• Radar
• IRST
• UW Acou.
RASSP Methodology
Specific Instantiation of Model Year Architecture

Figure 2 - 1: Model-year functional architecture elements

Section 3.0 of this document, the Model Year Architecture Technology Overview, introduces the concept of the functional architecture and the terminology of the Model Year Architecture (MYA). It discusses both the hardware and software aspects of the MYA.

Section 4.0 discusses the Implementation of the Functional Architecture. The Standard Virtual Interface (SVI) and Reconfigurable Network Interface (RNI) are introduced. These are the primary structures for supporting the model year hardware architecture for nodes and interconnect fabrics. References to the guidelines developed for example encapsulations are given.

An Overview of Emerging Interface Standards is presented in Section 5.0. The relationship of existing standards to the Model Year Architecture framework is discussed. Examples of the various types of interfaces are provided.

Section 6.0 presents, the Model Year Software Architecture. An approach to software layering which is used to achieve software application portability is described. Application programming interfaces are introduced which enable portability of applications both to new operating systems as well as new processors with new primitive libraries.

Finally, in Section 7.0, Implementation of Model Year in a Reuse System, the capabilities of a Reuse system that supports verification of a model-year architecture signal processor are briefly described. These are directly related to the most important aspects of creating, maintaining and using a reuse library, i.e., the existence of reuse elements at various levels of the modeling abstraction and the ability to recall reuse elements. The reuse library must support models at various levels of abstraction and elements in the library must be amply described in order to facilitate reuse.

RASSP Model Year Architecture (MYA) Appnote

3.0 Model Year Functional Architecture Technology

The technology overview provides a brief description of the MYA and its components. The MYA terminology and components are introduced.

3.1 Model - Year Architecture Functional Architecture

The functional architecture defines the necessary components and their interfaces to ensure that users can easily and affordably upgrade designs and insert new technology. The functional architecture is a starting point for users to develop solutions for an application-specific set of problems, not a detailed instantiation of an architecture. The functional architecture specifies:

- A high-level starting point to select application-specific architectures
- A standard approach to selecting and implementing standard, open interfaces
- Guidelines for efficient verification and test

The functional architecture **does not** specify the topology or configuration of the architecture, specific processor types, or system-level interface standards (external to the signal processor).

The functional architecture concept is based on the use of abstract architectural objects and standard functional interfaces at key points within a layered architecture, as shown in Figure 3 - 1. The key aspect of the functional architecture is ATL's approach to implementing the interfaces, particularly the various signal processor components (general-purpose processors, DSPs, special-purpose processors, and hardware accelerators), shared memories, sensors, and subsystem components (ancillary equipment, mass storage devices, etc.). The approach combines two concepts:

1. architectural layering
2. the use of standard, technology-independent functional interfaces

The primary reason for using a layered approach is that it provides logical decomposition into smaller, more manageable, understandable, reusable, and maintainable parts. The layered approach minimizes and confines changes that are introduced as a result of modifications (e.g., upgrades). A technology-independent functional interface is one that remains at the logical level, specifying no physical or electrical characteristics.

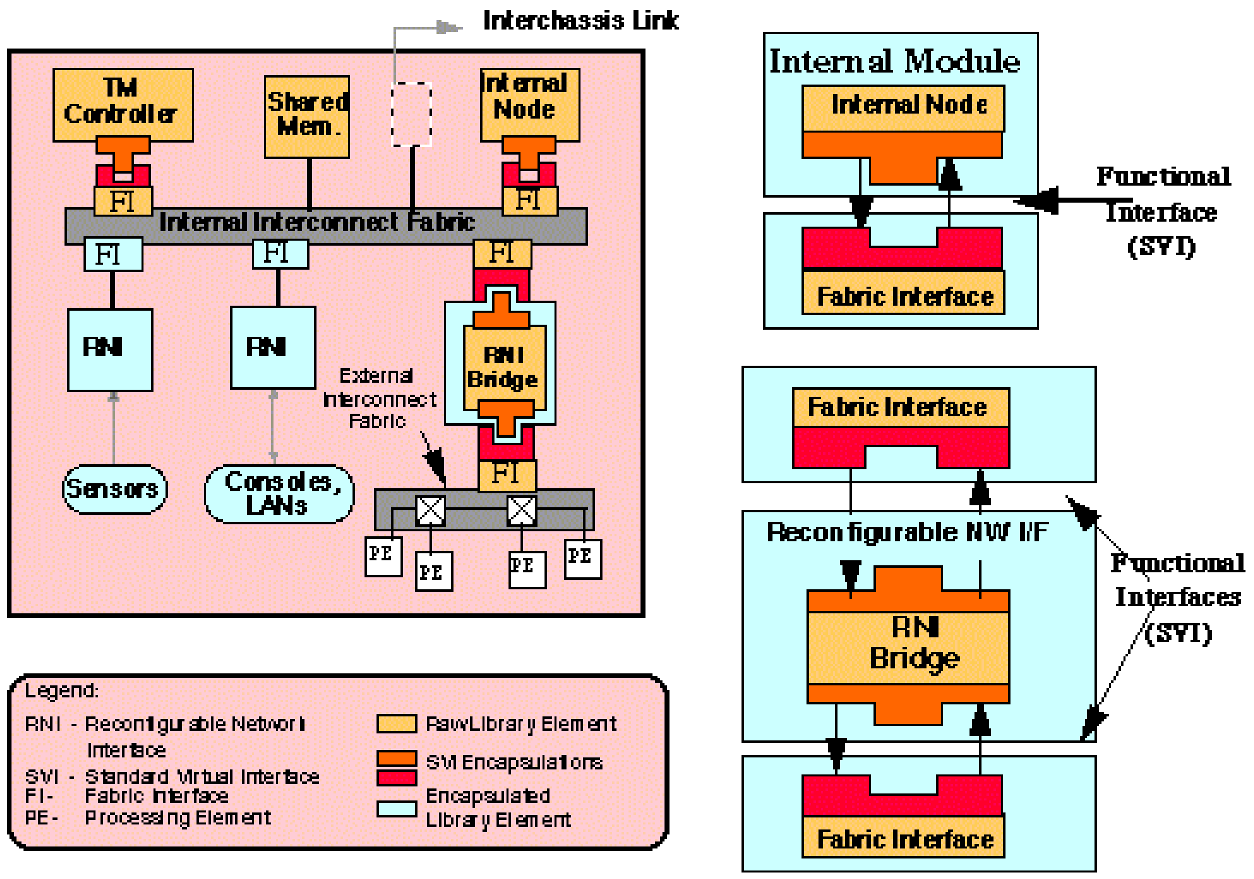

Figure 3 - 1: Model-year functional hardware architecture

At the hardware level, this is achieved by defining a VHDL encapsulation wrapper for each architectural reuse library element that implements a standard functional interface. An encapsulation wrapper is additional structure added to otherwise raw library elements to support the functional architecture and to ensure library element interoperability and technology independence to the maximum extent possible. The wrapper implements the hardware portion of the functional interface, called the Standard Virtual Interface (SVI), which is detailed in Section 4. It encapsulates the library element to hide implementation details, providing only an interface definition (the functional interface) to users.

3.2 Model - Year Architecture Software Architecture

The functional architecture includes both a software and hardware portion. The model-year software architecture simplifies developing high-performance, real-time DSP applications, allowing developers to easily describe, implement, and control signal processing applications for multi-processor implementations. As shown in Figure 3 - 2, it includes standard application programming interfaces (APIs) to the operating system and operating system services, and standard domain primitives for autocode software generation.

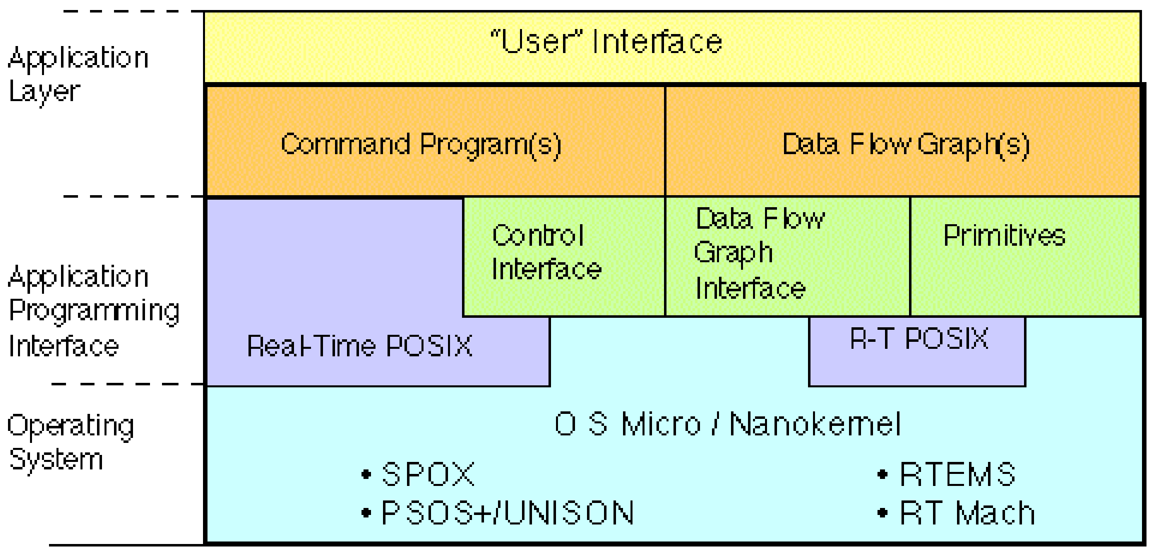

Figure 3 - 2: Top-level software architecture

The signal processing algorithms are represented as Data Flow Graphs (DFGs) that drive the autocode generation to produce the executable code for each processor in the system This process is supported by a reusable primitive library. The command program is the control flow program that provides the overall control, as dictated by the received messages.

The reusable run-time system and its support for the API, represented by the Control and Data flow Interfaces is Figure 3 - 2, is the essential component of software encapsulation/interoperability for a processor object. It provides the reusable data flow control and graph management for the signal processor. It is built upon a set of operating system services provided by a real-time microkernel using a standard interface. This system is a set of run-time application programs that constitute a 'virtual' machine which performs all graph execution functions and all system interface, control, and monitoring functions. It is usable with any operating system microkernel that provides the necessary set of services.

The application layer of the software architecture consists of an interface to the world outside the signal processor in addition to a command program and data flow graphs. The outside "user" could either be an actual user, or more likely, it could be a higher-level system as part of an overall platform. The division of the application into two parts is similar in concept to the Processing Graph Method (PGM) developed by the Naval Research Laboratory. The first part of the application layer is the command program, which provides response to external control inputs, starts and stops data flow graphs, manages I/O devices, monitors flow graph execution and performance, starts other command programs, and sets flow graph parameters. The control interface provides services that implement these operations. The second part of the application layer is the DFGs, implemented using a tool such as Management Communication and Control, Inc. (MCCI's) autocode or the LM/ATL's Graphical Entry Distributed Application Environment (GEDAE ™). The DFGs represent the signal processing algorithms that must be applied for each operating mode of the signal processor. Services provided by the data flow graph interface, which are largely invisible to the developer, include managing graph queues, interprocessor communication, and scheduling.

The constructed flow graph is converted via autocode generation into a higher order language (HOL), such as C or ADA containing calls to a standard set of domain primitives. The domain primitives are then mapped to optimized primitives in the target processor's library. Implementing the standard operating system APIs and standard domain primitive mapping via a Target Processor Map (TPM) forms the software encapsulation, which hides implementation details of the operating system, its services, and application libraries.

An important aspect of the functional architecture is that application-specific realizations of a signal processor are embodied in the proper definition and use of encapsulated library elements. Designers can use application notes in the reuse libraries to properly apply and aggregate individual hardware and software components into a final processor product. Implementing the object-oriented reuse libraries is described in Section 7.

The model-year architecture also provides a set of design guidelines and constraints to develop general architectures, such as how to properly use the functional architecture framework, general use of encapsulated libraries, and most importantly, procedures and templates to encapsulate new library components. The *Model-Year Architecture Specification Volume II - Hardware Architecture Element Specification* describes all elements of the functional architecture including the constraints.

While the hardware design tools to support the hardware functional architecture are already commercially available (i.e.

RASSP Model Year Architecture (MYA) Appnote

4.0 Implementation of the Functional Architecture Standard Virtual Interface (SVI) and Reconfigurable Network Interface (RNI)

4.1 Model-Year Hardware Interoperability

This section describes the levels of interoperability offered within the Model Year Architecture framework, and introduces the concept of encapsulation. Figure 4 - 1 shows the two specific levels of interoperable interfaces that are supported by the model year hardware functional architecture:

- internal node-to-internal interconnect fabric or node-level
- internal interconnect fabric -to-external interconnect fabric or interconnect-level. This interconnect-level may be simple, such as a connection between a slave peripheral and a system. It may also be quite complex, requiring protocol translation and data formatting services, such as a connection between heterogeneous networks.

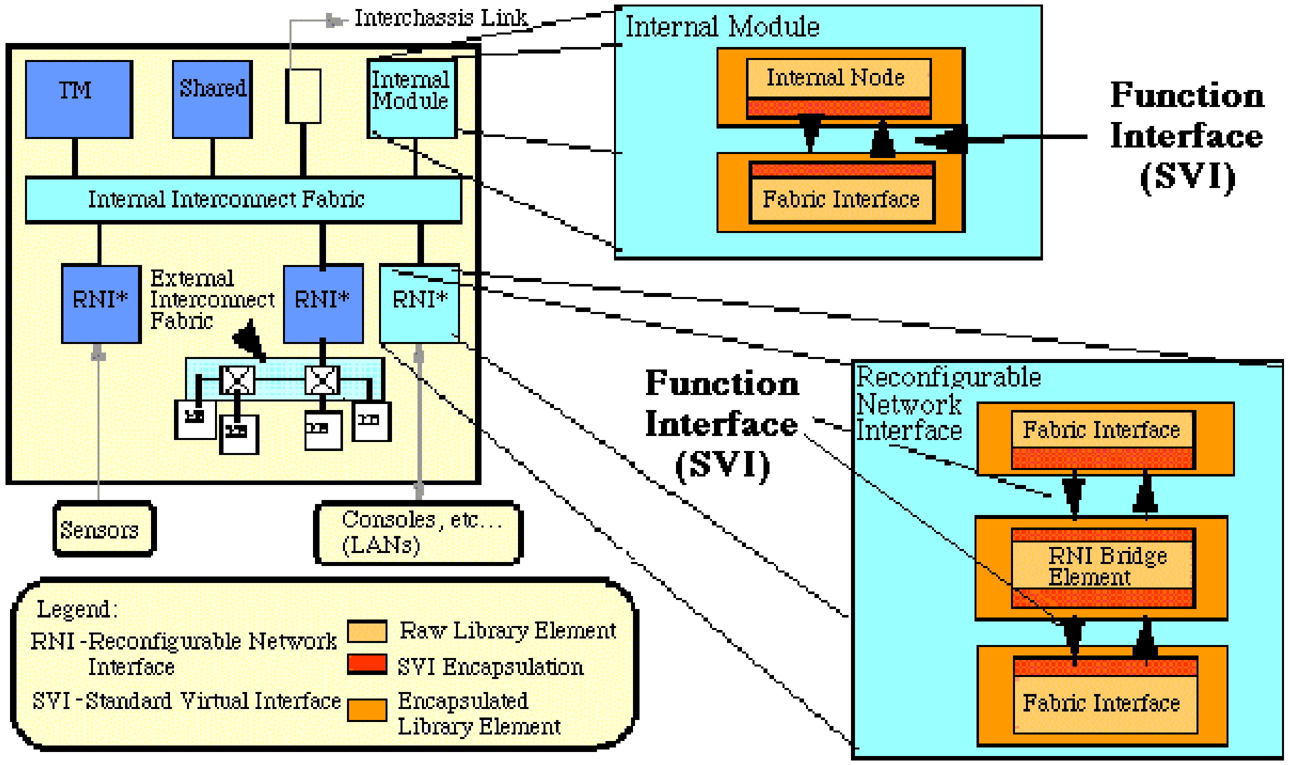

Figure 4 - 1: Two specific interoperable interfaces that are key to the model-year hardware functional architecture

Users implement node-level interfaces by encapsulating elements using the SVI. The SVI encapsulation is a VHDL wrapper that ports processor or processor bus interfaces to a common functional interface. The SVI specification was developed as part of the RASSP model-year effort, and interoperability experiments were run by porting several processors to various buses. Discussions of these experiments are included in Section 4.4 with additional details linked in the references.

Interconnect-level interoperability is provided using a bridge node between internal and external interconnect fabrics called the Reconfigurable Network Interface node (RNI). An RNI node is constructed from two SVI encapsulations interconnected by a logical layer that ties the two interfaces together. These nodes are usually implemented using a dual-port memory or FIFO and programmable processor or embedded controller. The SVI and RNI are described in more detail in the following two subsections.

4.2 Standard Virtual Interface (SVI)

The SVI defines a construct which makes it easier for users to implement node-level interoperability and to upgrade various architectural-level reuse elements, such as processor nodes and interface elements, by defining a standard interface encapsulation procedure. The SVI is a critical element of the RASSP model-year architecture. It guides users in developing architecture elements that can be easily reused and upgraded. In the past, integrating an internal node, which may contain several processing elements on a processor bus, to a specific interconnect was performed as a specialized implementation; this point-to-point integration approach can lead to a total of N^2 possible integrations for N nodes and interconnects. The intent of the SVI is to provide a common functional interface for all of these integrations, greatly reducing the number of integrations required. Once a node has been integrated to the SVI standard, users can integrate it to any interconnect integrated to this standard. This approach provides a level of plug and play interoperability that has never before been realized. The SVI is a functional, not physical, interface specification that supports technology independence - the 'virtual' in SVI.

Figure 4 - 2 illustrates how the SVI is used to encapsulate elements of the functional architecture. An internal node can be a single processing element or cluster of elements connected as a single node to a processor bus. Examples of internal nodes include signal processors, vector processors, or shared memory. An interconnect fabric can be any type of interconnect. Examples of interconnect fabrics include XBAR-based, point-to-point interconnect networks, rings, and multidrop buses. Examples of networks include Ethernet, FDDI, Fiber Channel, 1553B, etc.

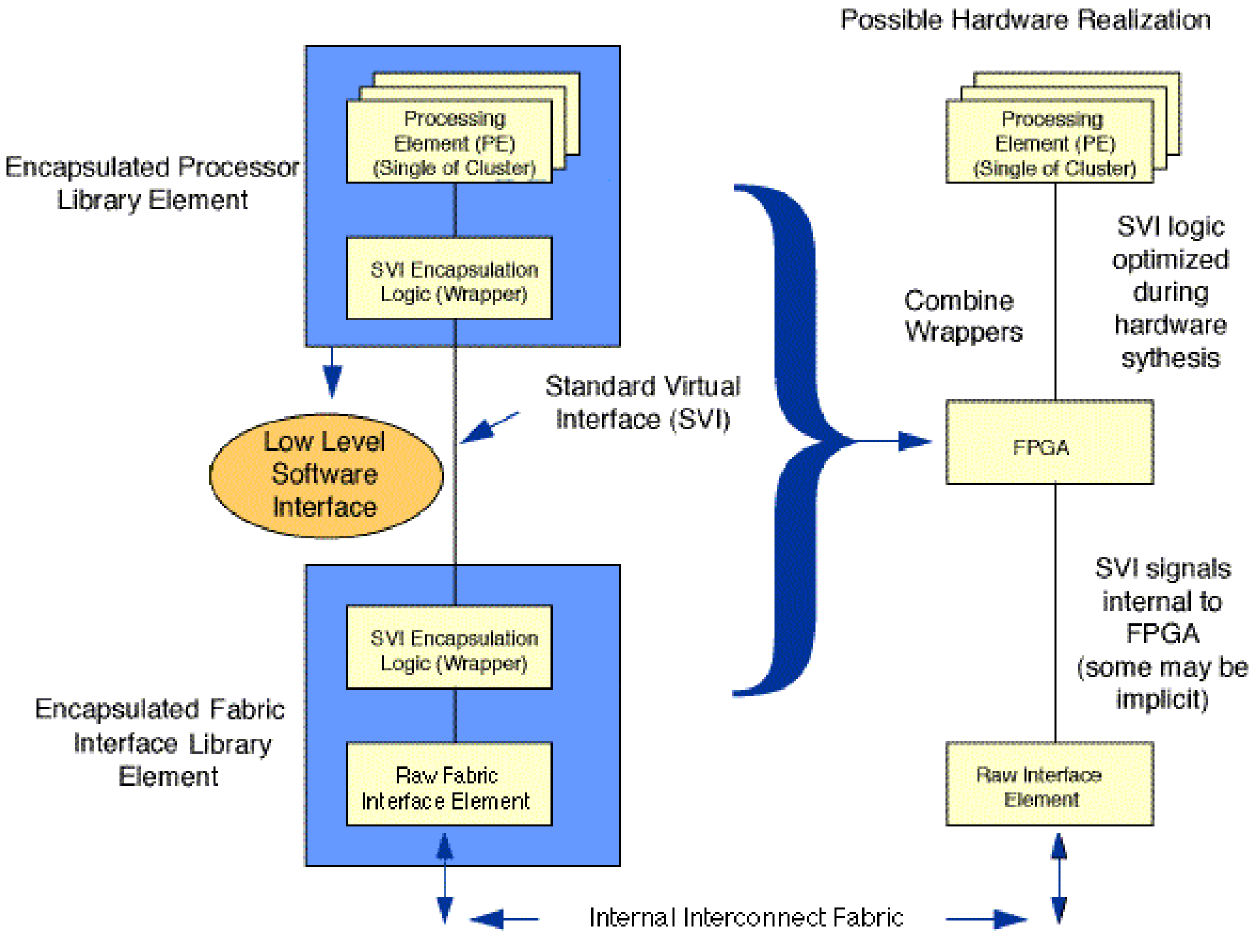
Possible Hardware Realization
Encapsulated Processor Library Element
Processing Element (PE) (Single or Cluster)
SVI Encapsulation Logic (Wrapper)
Low Level Software Interface
Standard Virtual Interface (SVI)
Encapsulated Fabric Interface Library Element
SVI Encapsulation Logic (Wrapper)
Raw Fabric Interface Element
Processing Element (PE) (Single or Cluster)
Combine Wrappers
SVI logic optimized during hardware sythesis
FPGA
SVI signals internal to FPGA (some may be implicit)
Raw Interface Element
Internal Interconnect Fabric

Figure 4 - 2: Standard virtual interface approach using internal node to fabric interface example.

Each library element (node or fabric interface) includes a VHDL wrapper that implements the SVI encapsulation logic. During the hardware implementation, the logic described within the encapsulations from both sides of the SVI is combined and optimized using synthesis tools to the greatest extent possible, This process may cause some of the signals defined for the SVI to become implicit in the remaining logic. What remains of the SVI is embedded within an ASIC, gate array, or FPGA, and would not appear as explicit pins on a chip or interface connector.

The SVI definition is designed to be general enough to handle different interprocessor communication paradigms. Some interconnect networks support a message passing paradigm to interprocessor communication, while others support a global shared memory paradigm. In some cases there is synchronous operation between the internal node and the interconnect fabric, while in other cases the internal node and the interconnect fabric operate asynchronously. The SVI is by definition synchronous; that is, each word of SVI data is transferred synchronously with the SVI clock. Support for asynchronous operation between an internal node and the fabric interface must be handled by the SVI encapsulation logic.

As shown in Figure 4 - 3, the data interface is partitioned into two unidirectional data interfaces: the Data Input Interface which receives incoming SVI messages, and the Data Output Interface which transmits outgoing SVI messages. The data interfaces are implemented with a master/slave pair: the SVI master is a data source to the SVI Data Output Interface, while the SVI slave is a data sink to the Data Input Interface. The encapsulation of a typical architectural element therefore contains an SVI master/slave pair. The advantage of partitioning the data path in this way is that it supports interface architectures with independent, concurrent data passing in both directions across the SVI. A bi-directional data interface can be obtained by using both unidirectional data interfaces controlled by the SVI master and the SVI slave.

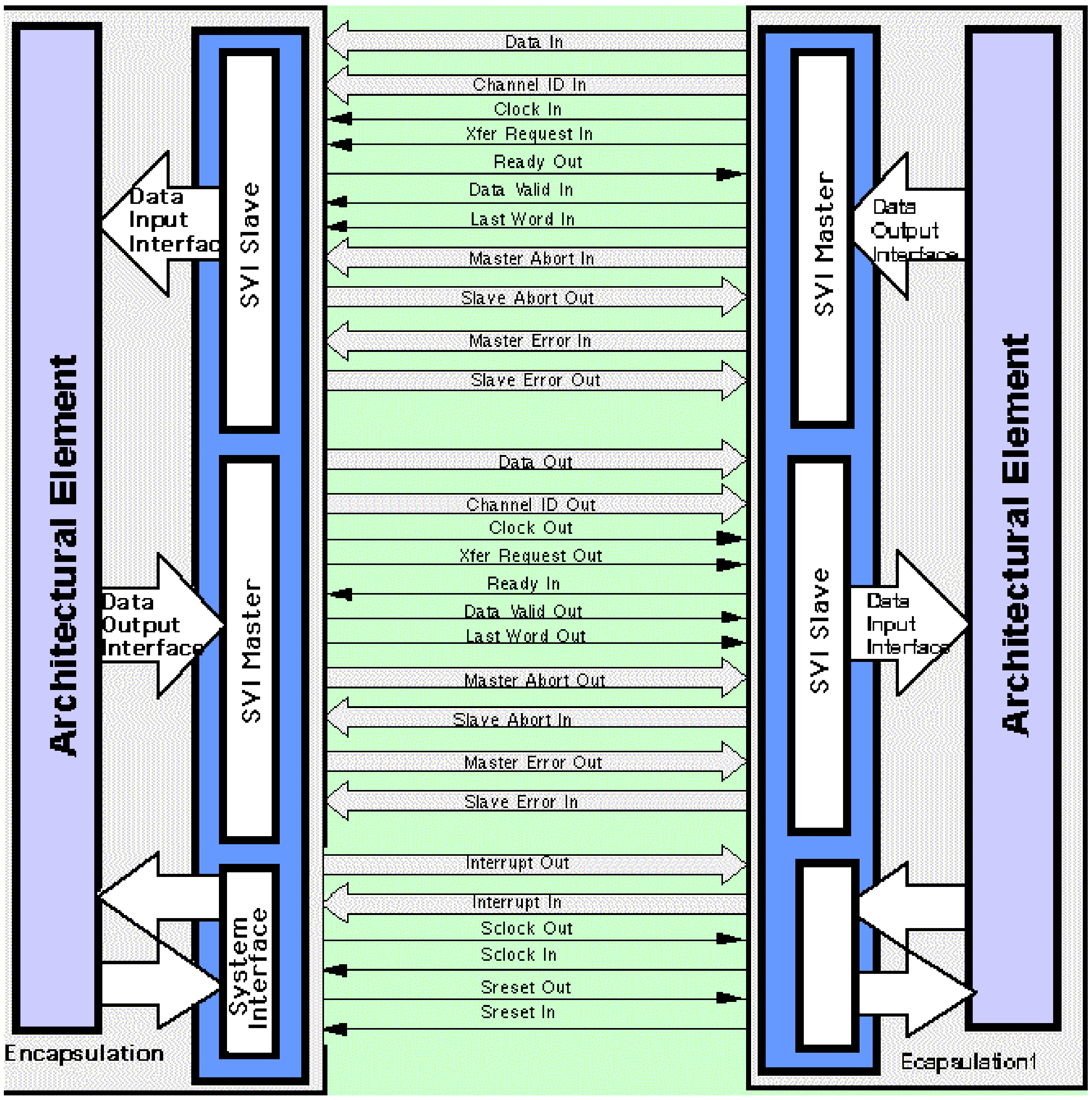

Figure 4 - 3: The Standard Virtual Interface

A set of SVI commands have been defined to implement the protocol. The SVI command is the first data word transferred across the SVI interface for each message. SVI data paths are implemented in byte increments. This allows encapsulations with different data-path widths. SVI commands occupy the least significant byte of the data bus, and transfer information across the SVI that will be needed by the receiving encapsulation. The address and data, along with any other necessary information, are in the data stream following the SVI command. The SVI master encodes the data in the SVI message and the slave decodes the message properly. The SVI message source must know how its encapsulation will build the message and also know how the receiving SVI encapsulation will interpret the data words in the message.

Typically, users encode the data in the message in software on the originating processing element. However, for low-latency operation, the message building can be done in hardware. The existing commands include those functions needed to implement SVI encapsulations envisioned to date; more commands may be necessary in the future to support new interconnects with unusual multi-processor or cache support requirements. Users can easily add commands, providing previous encapsulations with access to the new SVI commands.

To supplement the signal/timing/protocol definition and support end users" modeling efforts, the appendix to this application note contains VHDL code templates for SVI encapsulations. These templates include the entities and architectures a designer would typically need to code an encapsulation. In section 4.4, links to the VHDL code for several of the encapsulations performed as part of the verification study are provided. The code provided for the HOTLink encapsulation and the encapsulated RNI bridge are executable provided the reader has access to the Synopsys Logic Modeling's Smart Model library. To date, LM/ATL has implemented several SVI encapsulations. An interoperability demonstration has been developed in which two processing elements, a custom vector processor and several interconnect fabrics (PCI, RACEway, Myrinet and one sensor interface (HOTLink)) were encapsulated. Several combinations of the processing elements and interconnects were then mixed and matched to demonstrate interoperability. The results, as shown in Figure 4 - 4 for an SVI encapsulated PCI bus versus a custom interface, demonstrate the ability to provide plug and play interconnect between different elements with minimal performance impact. In this configuration, for data transfers of more than 256 elements, the performance impact is less than 5% of peak bandwidth.

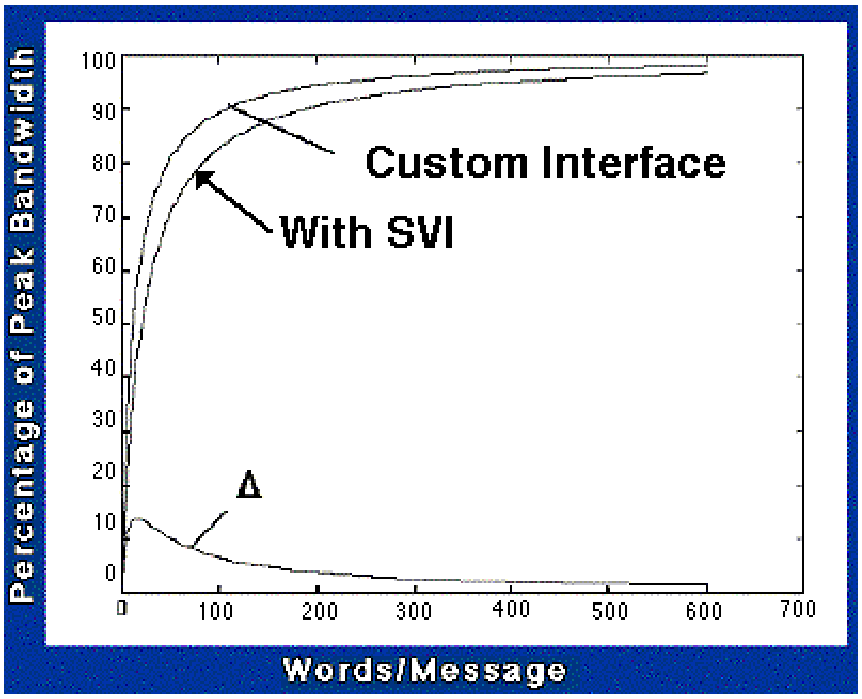

Figure 4 - 4: Performance Impact of SVI-encapsulated PCI vs. Custom Interface

In summary, the SVI is a critical element of the RASSP model-year architecture; it guides users in developing architecture elements that can be easily reused and upgraded. It must be used prudently, balancing the immediate performance impact against the future ability to upgrade to a new processing node, internal or external interconnect fabric.

4.3 Reconfigurable Network Interface

The RNI architecture shown in Figure 4 - 5 is the model-year architecture construct defined to support interconnect-level interoperability. The concept of formal separation of an internal or controlling node and the network or fabric interface, as defined by the SVI, is also applicable to the RNI. However, unlike the node-level interface, which connects nodes to buses or relatively simple interfaces, the RNI is a bridging element between heterogeneous interconnects. Another way of stating this is that the SVI supports interoperability for the lower ISO/OSI layers (layers 1 - 2 physical/datalink), while the RNI supports interoperability for the higher layers (layers 3 upward Transport+up). A specific example of this might be a bridge between the Scaleable Coherent Interface (SCI) and a MIL - STD - 1553B network that interfaces to a navigation subsystem.

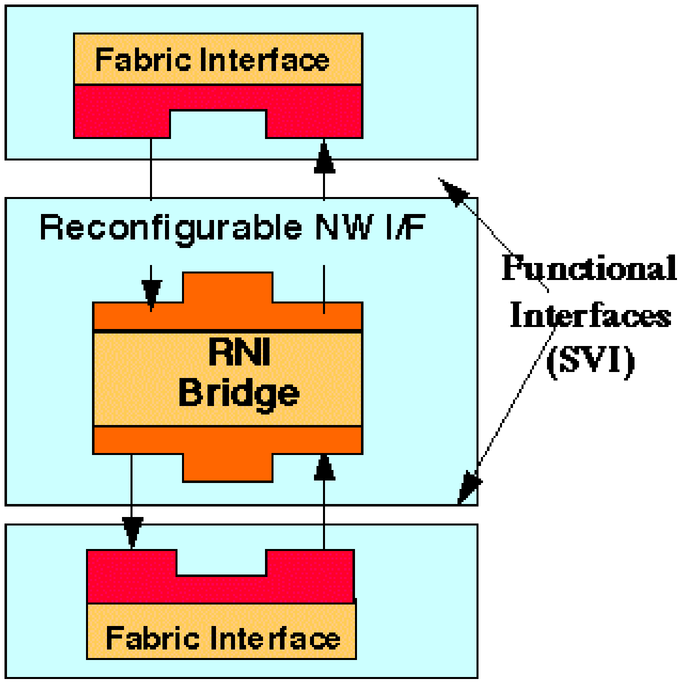

Figure 4 - 5: Reconfigurable Network Interface Architecture

The RNI consists of 2 encapsulated fabric interfaces on either side of an encapsulated bridge element. The fabric interfaces implement the two specific interfaces to the interconnect fabrics. The bridge may be between an internal and an external interconnect fabric or a it may be between two external interconnect fabrics. The actual bridging function is performed by the RNI bridge element, which consists of a buffer memory to facilitate asynchronous coupling between the two interfaces, and a controller which coordinates data transfers and provides flow control. The bridge element can be implemented via custom logic (e.g. FPGA, ASIC) or a programmable processor.

As in the case of Internal Module interface, a layered communication approach is employed. Since the RNI provides a bridge between two interfaces, two separate OSI-layered structures exist at each interface. The lowest layers of the two protocols are translated in the Fabric Interface components. The higher levels of each protocol are translated within the RNI bridge element and converge when the data interchange is stripped of all its interface specific identity, and exists as a pure SVI message on the SVI protocol. Again, the ISO/OSI model is being used as a reference and does not imply that the layering ultimately employed in defining the internal node architecture will strictly follow this model or implement all seven layers. An example of an encapsulated RNI bridge is described in [Myri_PCI]. This model is executable if the user has access to the Synopsys Logic Modeling's SmartModel Library.

The exact implementation of the RNI itself depends on the type of interfaces being served. The RNI to implement interfaces for loosely coupled processing subsystems, operator consoles, and certain types of ancillary equipment will have one implementation, while the RNI to support a remote sensor application will have another. As mentioned above, the complexity of the data transfer protocol may also have a wide variance. A sensor or multidrop bus would generally require a fairly simple protocol supported by a relatively simple hardware controller within the RNI bridge element. This type of RNI is referred to as a simple RNI. On the other hand, a local area network application most likely would require a programmable processor to implement a more complex software-based protocol. This type of RNI is referred to as a complex RNI. More than one class of RNI will be required to support all aspects of signal processor interfacing. As described in Section 7, the RNI will be supported through various object subclasses or derived classes for Fabric Interfaces, both internal and external, and RNI Bridge Elements, within the reuse library.

4.4 MYA Application Example

4.4.1 Introduction

Several encapsulations were done for the Model Year Architecture Hardware Verification effort. These include encapsulations of internal nodes, multidrop buses, sensors and networks. First, how to determine which architectural interfaces to encapsulate will be discussed. Following that there is a short description of the encapsulations which have been performed at LM/ATL. At the end of each description is a link to the complete application note for each encapsulation.

4.4.2 Approach

To apply the Model Year Architecture, the designer performs the following steps:

- Select the high payoff interfaces for encapsulation
- Research existing encapsulations to find examples that are similar to new design
- If a similar design is found, modify it to meet new requirements. If not, start with templates.
- Develop and validate a virtual prototype

4.4.2.1 Select the high payoff interfaces for encapsulation

The first key step in applying the Model Year architecture is to take the candidate implementation and evaluate the different types of communications interfaces of the Functional Architecture to determine: what interfaces are at the periphery of modules that will require regular upgrading?

There are basically three places to look for upgradeable interfaces as shown in Figure 4 - 6.

- an internal node or its associated internal interconnect fabric
- an external interconnect fabric
- an external device (peripheral)

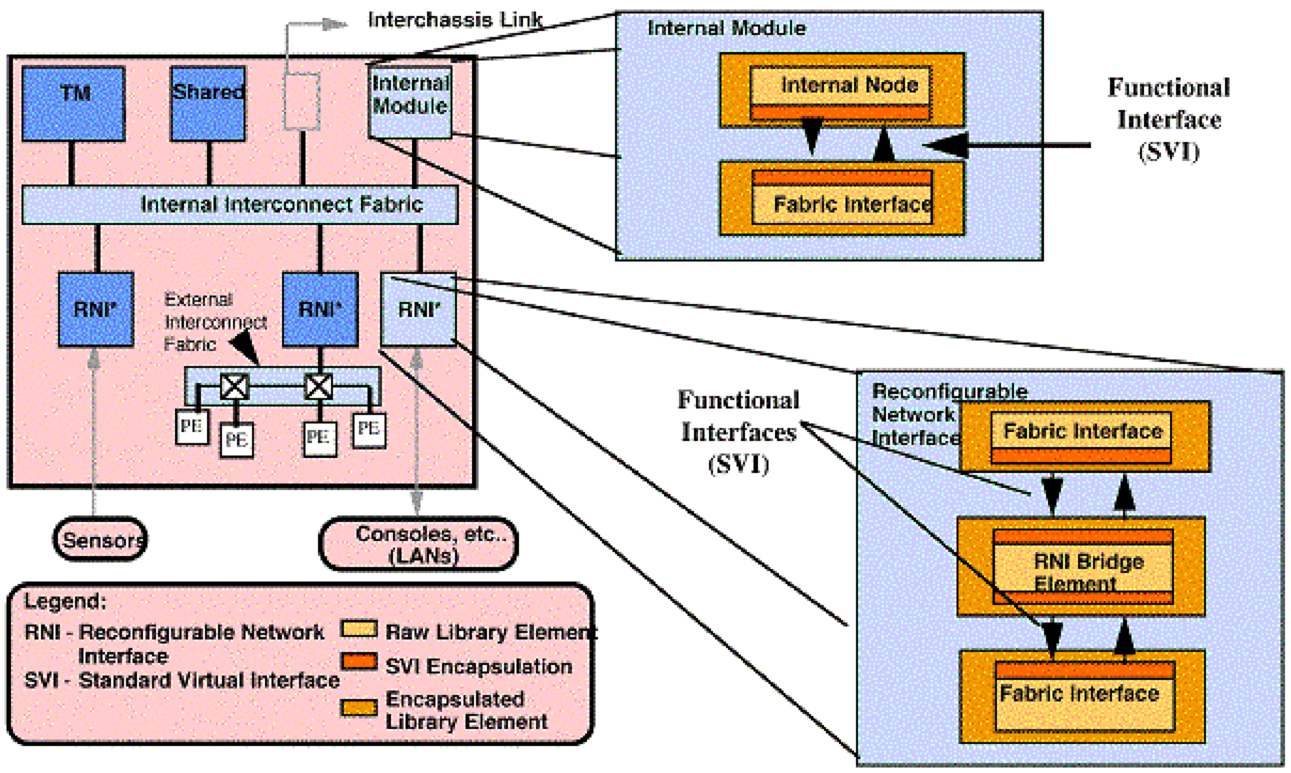
Interchassis Link
Internal Module
TM
Shared
Internal Module
Internal Node
Fabric Interface
Functional Interface (SVI)
Internal Interconnect Fabric
RNI*
External Interconnect Fabric
RNI*
RNI*
PE
PE
PE
PE
Functional Interfaces (SVI)
Reconfigurable Network Interface
Fabric Interface
RNI Bridge Element
Fabric Interface
Sensors
Consoles, etc.. (LANs)
Legend:
RNI - Reconfigurable Network Interface
SVI - Standard Virtual Interface
Raw Library Element
SVI Encapsulation
Encapsulated Library Element

Figure 4 - 6: Functional Architecture

4.4.2.2 Research existing encapsulations to find examples that are similar to new design

For the Model Year Architecture implementation, the first step would be to search the Reuse Library for available encapsulations. Through the course of the verification studies, it has been found that the process of designing a new encapsulation is simplified by modifying or at least referring to a similar existing one. There are architectures and portions of architectures, such as case structures in existing encapsulations, that are often able to be used again with only slight modifications. The following sections describe the different types of encapsulations and links to available code. Not all of the encapsulations can be executed by the reader due to the proprietary nature of some of the underlying models. Nevertheless, portions of the model may be reusable.

4.4.2.2.1 Internal Node and its associated processor bus

An internal node is any architectural-level element which becomes connected (through a fabric interface) to the internal interconnect fabric. Examples are signal processors, vector processors or shared memory. In Figure 4 - 7, the internal node consists of 4 Wafer Scale Signal Processors (WSSP) interconnected on a proprietary bus called the IOBUS. Encapsulating the IOBUS is equivalent to encapsulating the Internal Node. This encapsulation is not available due to the proprietary nature of the IOBUS.

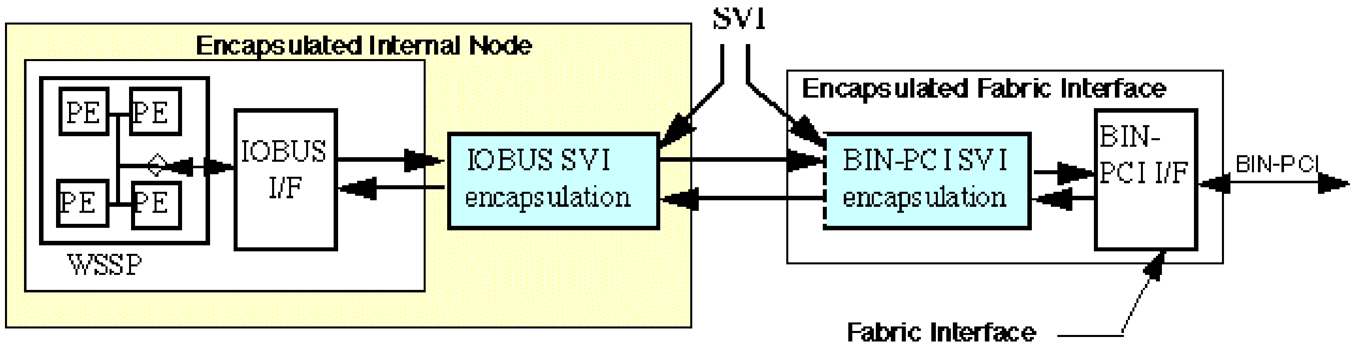

Figure 4 - 7: Encapsulated Internal Node

4.4.2.2.2 Fabric Interface

The fabric interface is an element which controls the transfer of data between attached nodes and internal or external interconnect fabrics.

- A bus controller for a PCI-like bus called the Board Interconnect Network (BIN)-PCI bus on the WSSP program was encapsulated as shown in . This encapsulation is described in [PCI_SVI] (Note: BIN-PCI is a special PCI bus implementation done by the Air Force Rome Laboratory)).

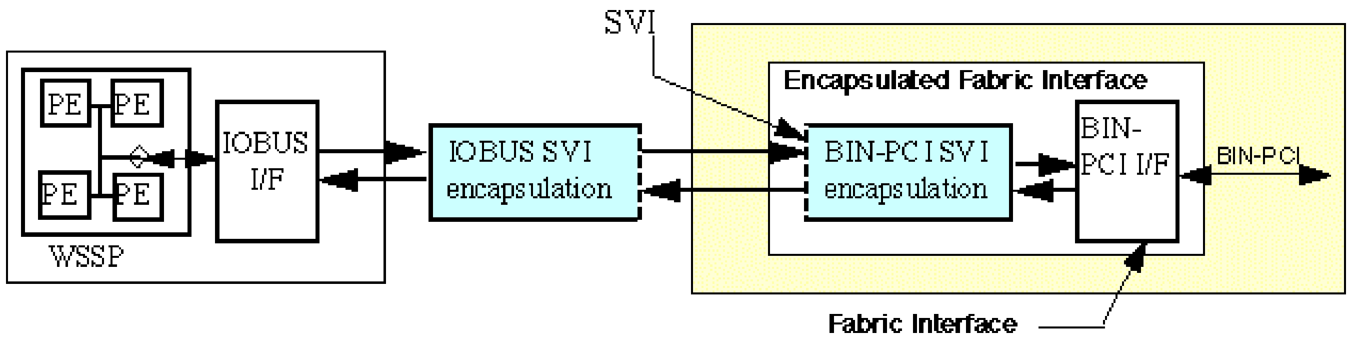

Figure 4 - 8: Encapsulated Fabric Interface

- The VHDL for the encapsulation is not available due to its proprietary nature. A discussion of the hardware synthesis of the IOBUS to BIN-PCI SVI interface encapsulations is available in [PCI_SVI].

- The Mercury RACEway network was encapsulated so that the RACEway link to other processor subsystems and interfaces would be upgradeable. Similarly, the Myrinet fabric was encapsulated. These two networks were then interconnected via a programmable bridge to form a complex RNI. The two encapsulations of the RACEway and Myrinet networks are described in [RACE_SVI] and [Myri_SVI].

- A complex RNI bridge was developed to interconnect the Myrinet to the PCI. This example is a good template for the bridge element for complex RNI encapsulations since the bridge programming is fairly generic. This is described in detail in [Myri_PCI]. This bridge is the same bridge architecture that is used to interconnect the RACEway and the Myrinet.

4.4.2.2.3 Sensor Interface

The sensor interface is a key element that should be encapsulated. This is an entity that is likely to be upgraded and/or reused.

- The HOTlink high speed serial port is an example of a sensor interface encapsulation. The HOTlink encapsulation interconnected through a simple RNI bridge to the encapsulated PCI bus is described in [Hotlink_PCI].
- The Data I/O bus from Benchmark 2 SAR processor was encapsulated and interconnected to an encapsulation of Mercury"s RINO chip set. This would support an upgrade of either the Hotrod interface on the Data I/O board or the RACEway network to which the RINO interfaces. This is described in [Data_IO_RINO].

4.4.2.3 Use of templates

If a similar design is found, modify it to meet new requirements. If not, start with templates. The templates contain entities and architectures for the SVI. They lend structure to the encapsulations, so that even though different people may design the encapsulations, they will all look somewhat the same, making them more easily reusable. They are available at [SVI-Template]

4.4.2.4 Develop and validate the Virtual Prototype Implementation

The tools used to support this detailed design and synthesis by the RASSP team include the following. Other similar vendor tools may be substituted and should achieve the same results.

- Visual HDL - to capture the VHDL
- QuickVHDL - to simulate and validate the design
- Synopsys - to synthesize the design implementation
- Various other hardware tools to support FPGA or ASIC - based implementation (Refer to the CAD System Description document for the complete list of tools used in the RASSP program)

4.5 Validation Techniques

It is often difficult, on the one hand, to generate data in the format required by some protocols and, on the other hand, to interpret data generated by some protocols. To alleviate this problem, LM/ATL has used the encapsulations to input and output the data. That is, once an encapsulation is completed, the slave encapsulation can be used to supply data to the model and the master to extract data from the model. Normally, the slave translates the SVI protocol into the interface protocol. Referring to Figure 4 - 9, it is clear how the slave can be used to supply data to the network and the master, to extract it. The user can supply an SVI data set which is very easy to develop, to any encapsulation, and also have the output supplied in the easily readable SVI format

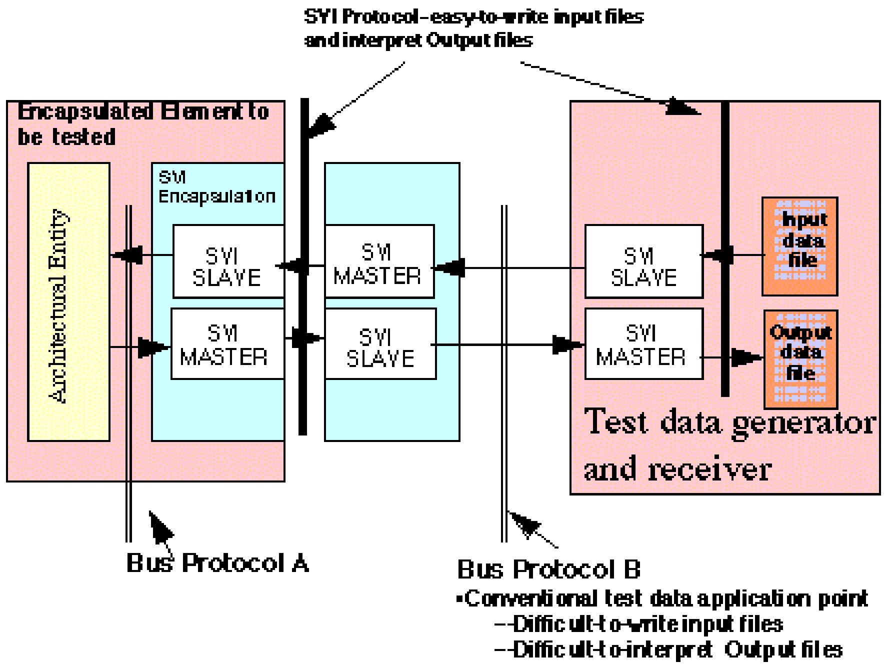
SVI Protocol–easy-to-write input files and interpret Output files
Encapsulated Element to be tested
Architectural Entity
SVI Encapsulation
SVI SLAVE
SVI MASTER
SVI MASTER
SVI SLAVE
SVI SLAVE
SVI MASTER
Input data file
Output data file
Test data generator and receiver
Bus Protocol A
Bus Protocol B
•Conventional test data application point
--Difficult-to-write input files
--Difficult-to-interpret Output files

Figure 4 - 9: The encapsulated element can be used to generate and receive test data and results

4.6 Development Time

The difference in developing the hardware for a MYA compliant interface compared to developing the hardware for a custom interface has two aspects. The first is the actual time it takes to develop the interface. In the case of the MYA, the first time an interface is designed, it is necessary to encapsulate either an internal node or fabric interface and whatever that node or fabric interface is connected to. This will generally take longer than a custom interface, since two interfaces are being designed.

The savings comes when the internal node or the fabric interface is upgraded, which is the second aspect of the development time. For example, assume there exists an encapsulated internal node and an encapsulated fabric interface, and that the designer is upgrading the internal node. The encapsulated fabric interface remains essentially the same, and the new node is encapsulated. It is anticipated that this will take less time than doing another custom interface because it is not necessary to write new drivers for the fabric interface. It is also not necessary to learn the protocol of the fabric interface. Furthermore, once the new internal node is encapsulated, it can be interfaced to any other encapsulated fabric interface. If there are 10 encapsulated interfaces, one needs only to encapsulate the internal module once to interface it to all of the fabric interfaces. If all of the interfaces were custom interfaces, one would need to design 10 different custom interfaces to achieve the same result. Clearly, this reduction in the number of integrations required from the order of N^2 to N equates to a significant savings in development time.

Next Up Previous Contents
Next: 5 Emerging Interface Standards Overview **Up:** Appnotes Index **Previous:** 3 Model Year Functional Architecture Technology

Approved for Public Release; Distribution Unlimited *Dennis Basara*

RASSP Model Year Architecture (MYA) Appnote

5.0 Emerging Interface Standards Overview

5.1 Interface Standards

Open interface standards should be used within systems wherever possible to further ensure interoperability between components. Using commercially accepted and non-proprietary standards may preclude the necessity of functional interface encapsulations in some cases. This is bedause COTS fabric interface components may allow direct communication between architectural elements and the selected standard interface. Confining designs to open interface standards will:

- ensure multiple sources and reasonable costs for compatible components
- eliminate dependence on sole source proprietary components
- ensure a well-understood and problem-free interconnect technology due to its acceptance and proliferation elsewhere in the military and/or industry.

It was the intention at the time of the writing of the RASSP MYA Working Document that the scope of the Model Year Architecture would include the maintenance of a set of approved RASSP accepted and recommended open interface standards. Upon further consideration it was determined to be impractical and inappropriate for RASSP to attempt to qualify interface standards, but to instead provide a recommended methodology for selecting appropriate interface standards. There are several factors mitigating against RASSP-accepted interface standards. The primary issue is that there can be several applicable standards for a particular signal processing system application, one of which may be more appropriate for a particular application than another. It is not in the best interest - or the intention - of RASSP to dictate which standard is most appropriate. In addition, organizations such as the Navy's Next Generation Computer Resources (NGCR) and the Air Force Joint Integrated Avionics Working Group (JIAWG) have expended a great deal of time and effort in evaluating and selecting standards applicable to their respective application domains. There is little value added for RASSP to reinvent the results of these organizations. Finally, not all interfaces are under complete control of the signal processor design. For example, subsystem interfaces used to interface RASSP signal processors to a large amount of ancillary equipment and subsystems previously designed may be outside the control of the designer.

The RASSP Model Year Architecture Specification - v1.0 , Section 7, describes a process which may be used to systematically select the optimum interconnect approach for a particular design from among open interface standards. In addition, the VSIA is developing a standard way of documenting performance and physical information about standard buses. Used in conjunction with the RASSP process, this will provide a very useful tool for comparing interconnect standards.

Of all the standard interfaces, the test interface required extra consideration. During RASSP a Design-for-Test (DFT) Methodology was developed. This methodology requires the use of standard test interfaces. However, there are only a handful of standard test interfaces available, as described in the RASSP Model Year Architecture Specification - v1.0 , Section 7, and they are widely accepted and non-overlapping. Therefore, it did not make sense to specify a particular hardware functional test interface.

Another aspect of the test interface which was considered was the test Application Program Interface (API). That is, the programming interface to the test structures. In contrast to the functional hardware test interface, there is currently no standard test API. The reason is, the APIs that many vendors are supplying with their boards are very hardware specific. It is unclear whether a standard test API could be developed that would be efficient enough to be useful. The benefit of such an API is clear. As portions of the system are upgraded, the top level test software would not have to be completely rewritten, although it might be expanded or reduced depending on the capability of the upgrade. In order to fully support the RASSP architectural process, the API would have to be implementation independent. The new hardware or software would have to supply appropriate results to the API calls.

RASSP Model Year Architecture (MYA) Appnote

6.0 Model Year Software Architecture

6.1 Model-Year Software Architecture

The model-year software architecture and interoperability relies heavily on a standard API and services supplied by the run-time system, which encapsulates the operating system and the primitive libraries. The model-year software architecture is shown in Figure 6 - 1, along with the associated elements users need to automatically generate software. The intent on RASSP is to drive toward graph-based autocode generation using data flow graphs (DFGs) to represent application (algorithm) code and state machine diagram graphs to represent command/control code to the maximum extent. In the event of a processor upgrade, the graph-based representation provides the common definition of operation and service requirements even though the underlying processor hardware, operating system, and libraries have changed. The same holds true if a primitive library or operating system for a given processor type is upgraded.

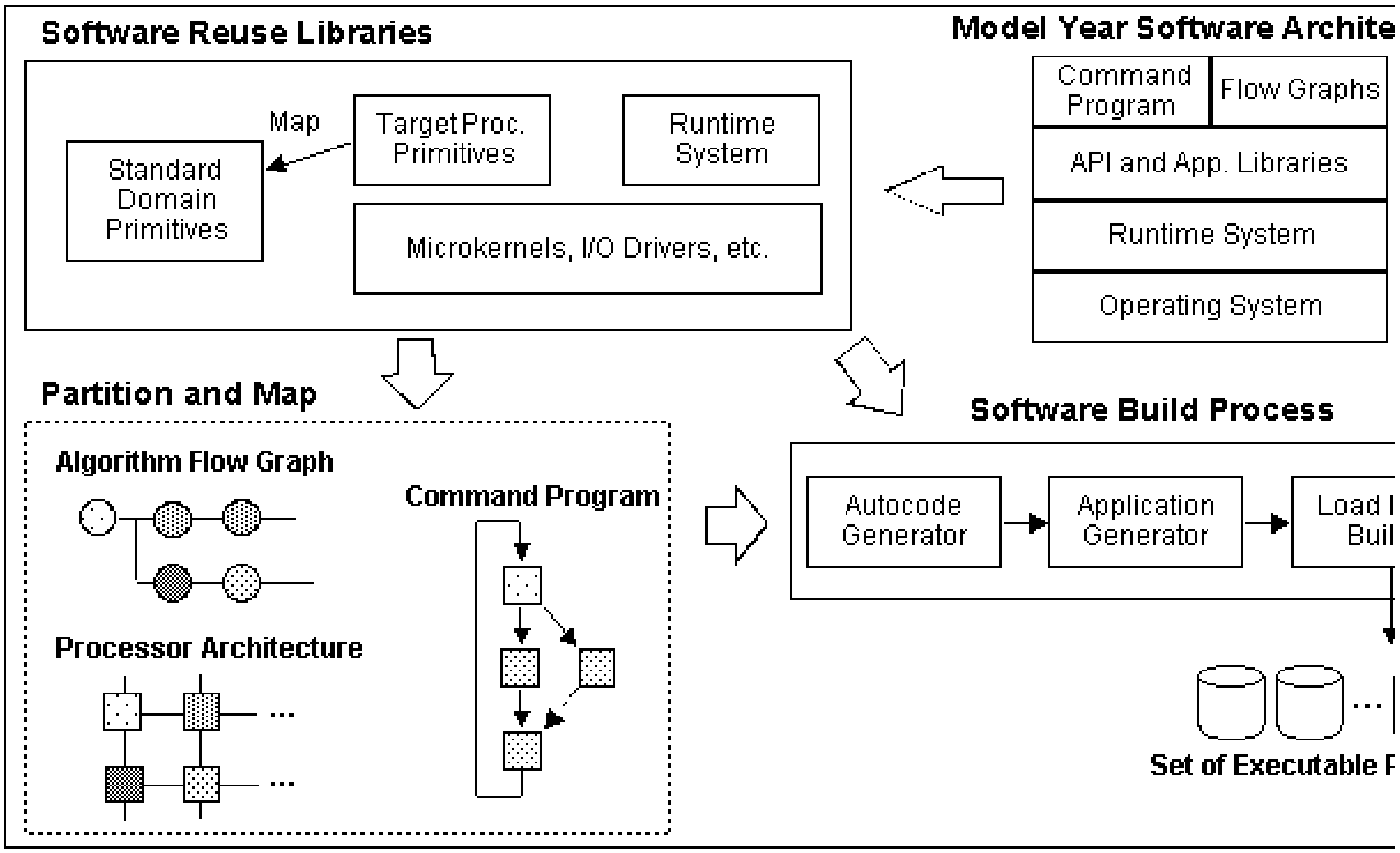

Figure 6 - 1: Model Year Software Development Environment

The model-year software application layer is divided into two parts: the command program and the application's data flow graphs. The command program does the following:

- responds to external control inputs
- starts and stops data flow graphs
- manages I/O devices
- monitors flow graph execution and performance

- starts other command programs
- sets flow graph parameters

The control interface provides the services that implement these operations as described in the Autocoding for DSP Control Application Note. Command software state machine descriptions that can be autocoded using such tools as Verilog's ObjectGeode were developed on the program. The command program can directly use the high-level interface to operating system services provided by the Real-Time POSIX compliant system. Building the command program based on a standard control interface and on POSIX isolates it from the microkernel and allows it to be easily ported to each new model-year architecture.

6.2 Application Software

An application's data flow graphs are developed using a data flow paradigm such as PGM's Signal Processing Graph Notation (SPGN) or GEDAE™. The DFG interface services are largely invisible to the developer, they include:

- managing graph queues
- interprocessor communication
- and scheduling node firing

In addition to the DFG interface, the DFG paradigm needs a set of primitive nodes that can be used to build the graph. A SPGN graph is converted into an Higher Order Language (HOL) , such as C or ADA, via autocode generation. The resulting code contains calls to a standard set of domain primitives. Typically, these primitives implement such operations as FFTs, filters, waveform generators, and matrix manipulations, among others. To ease portability to various processors and to take advantage of vendor and third-party-optimized target-processor libraries, Target Primitive Maps (TPMs) translate the standard domain-primitive calls to those specific to the desired support libraries. The TPM translates calling conventions of the domain primitives into those compatible with the target-processor libraries. This may also include translating a single call of a domain primitive into different target-processor library element calls, depending on a particular domain-primitive parameter or parameters. A simple example of this may be an FFT/IFFT domain primitive that might be mapped to separate FFT and IFFT target-processor library elements, depending on a forward/inverse switch in the domain-primitive parameters. Refer to the Hardware/Software Codesign and the Data Flow Graph Design Application Notes for a more detailed explanation of DFG and these interface services.

6.3 Operating System Service Requirements

Support for the Application Program Interface (API) is through a run-time system, which provides the services required to control and execute multiple graphs on a multi-processor system. The run-time system relies on support from the underlying operating system (microkernel and support services). The data flow graph approach proposed as a signal processing API does not imply that the associated scheduling and execution paradigms implemented by previous PGM run-time systems must also be used for the RASSP implementation.

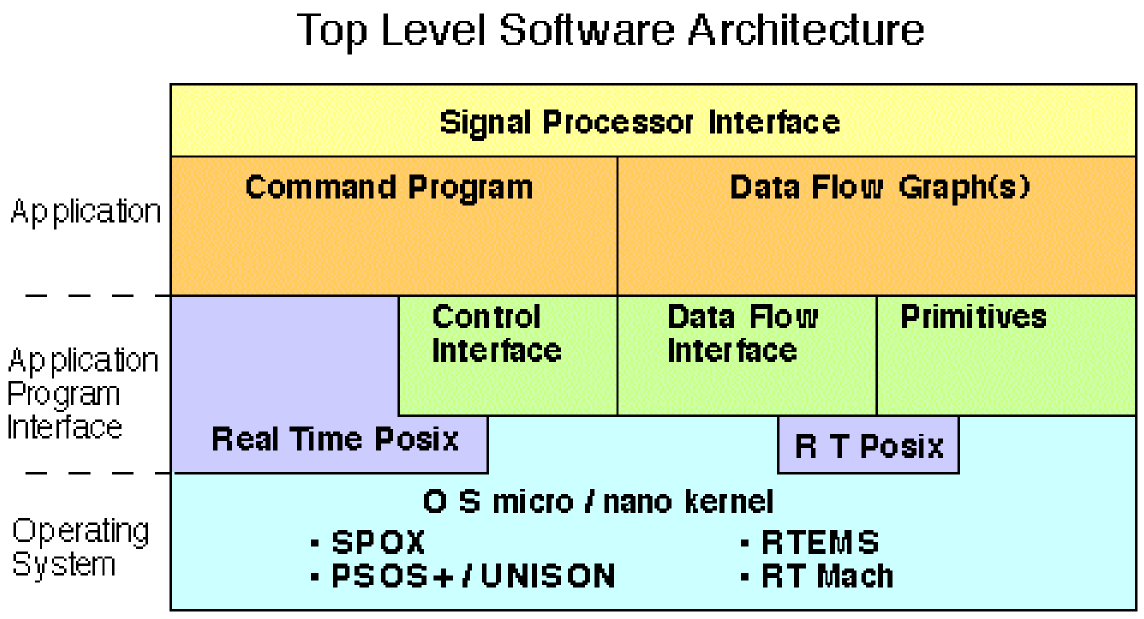

Figure 6 - 2: Top-level software architecture

The operating system services must support the run-time system. Both GEDAE™ and MCCI provide run-time systems that satisfy the RASSP requirements. As shown in Figure 6 - 2, the run-time system interfaces directly with the operating system for some required services. The interface is isolated to a low level to simplify portability; users can customize it for a particular operating system. The term operating system includes the microkernel itself, plus any external services not implemented directly within the microkernel that are needed to support the run-time system.

The operating system services must also provide guaranteed performance limits on the microkernel services. Five categories of service requirements must be provided:

- Task control using processes and threads
- Interrupt handling
- Memory management
- Interprocessor communication
- I/O services.

Table 6 - 1 details these service requirements and the primary performance and interface issues associated with each. These required services must be used to evaluate and select suitable microkernels for RASSP signal processors.

Table 6 - 1: Service requirements for microkernel

Requirements Category	**Services Provided**
1. Tasks control using processes and threads	- context witch time (process and thread) - take/give semaphore time (intra- and inter- process) - thread control: spawn, suspend, reume, terminate, priority - multiprocessor thread support
2. Interrupt Handling	- Interrupt response time affected by: - Kernel overhead in handling interrupt - Kernel disabling of interrupts for time T_{off}
3. Memory Management	- Performance of alloc and free - Class of memory - onchip, SRAM, cache, etc.
4. Interprocessor Communication	- Shared Memory - Transparent, i.e. shared globals - Explicit: interface for copying to/from shared memory - Point-to-point Communication - Buffer sizes/types - Amount of data copying required - Heterogeneous Processort (data translation standards) - XDR, Format description , etc. - Scalability
5. I/O	- Ability to add new device drivers - Ability to suport BIT and performance monitoring

The capabilities provided by these services must support typical embedded multiprocessor applications. These applications will be comprised of host and target processes, all running as peers, grouped on multiple processors. Multiple processes grouped on any single processor must be able to run in parallel. Since a group of processors may physically reside on the same or different boards, a common method of memory addressing is needed to eliminate memory overlaps and address ambiguity between processors. The operating system must have the characteristics of a distributed, preemptive, multi-processor, multi-tasking operating system, while also supporting the high-performance throughput capabilities of tightly coupled processes. The operating system will provide a kernel-level interface to the multi-tasking preemptive kernel. This interface will include allocation and control functions for local memory, functions to connect and control interrupt handlers, and functions to manipulate programmable hardware that may be on various processor systems, such as DMA controllers and timers.

The RASSP approach can accommodate signal processor designs that include COTS products with a proprietary operating system as long as the operating system meets the service requirements defined for RASSP, and as long as it provides an open interface on which the run-time system and API(s) can be ported. The important issue, as with all RASSP designs, is to ensure that implementing the underlying operating system, its services, and API is transparent to the application software to make it easier to insert model year upgrades of hardware and system-level software.

Next Up Previous Contents

Next: 7 Implementation of a Model Year in a Reuse System **Up:** Appnotes Index **Previous:**5 Emerging Interface Standards Overview

RASSP Model Year Architecture (MYA) Appnote

7.0 Implementation of a Model Year in a Reuse System

7.1 Library Reuse Elements

LM/ATL's RASSP team has developed a formal definition of reuse library development by extending the concept of object-oriented design (which has been successfully applied to software development) to signal processor architecture design, encompassing both hardware and software. Architectural library elements, such as a processor element, can be described as:

- an abstract entity defined by an object class which possesses a standard interface definition, with
- a defined functional capability or set of methods implemented via a combination of hardware and software.

Architectural reuse elements are written at a level of abstraction to hide hardware and software implementation details from users and limit the impact of design changes.

Implementing the object's functionality requires a co-dependency of its hardware and software aspects, necessitating **hardware-software codesign**. For example, operating systems must be configured to take advantage of whatever hardware resources are available to support its functionality. Likewise, application libraries may need to be configured or optimized to take advantage of specific hardware implementations.

7.2 Model Hierarchy

Users iteratively verify a model-year architecture signal processor throughout the codesign process; they require the reuse libraries to support models at various levels of hierarchy. The ATL team has worked with models at four levels of the VHDL modeling hierarchy

- Performance Models (synonym: Uninterpreted Model) provide timing-only and control functionality for processor nodes buses/interconnects, etc. to support high-level architectural trade-offs (number and types of processors, type and topology of network)
- Abstract Behavioral Models provide behavior at the data output level with (potentially) some level of timing. This level includes both algorithm-level and Instruction Set Architecture (ISA)-level models.
- Interface Models (synonym: Bus-functional) describe the operation of a component with respect to its surrounding environment.
- Detailed Behavioral Models, previously known as Full-functional models, describe functionality at the signal level and timing fidelity at the clock level. This includes Register Transfer Level (RTL) and logic models.

These models are more fully defined in the RASSP Terminology and Taxonomy document and the Token-Based Performance Modeling application note. Through these models, the functional architecture constructs are supported. The use of these models has been tested in a series of benchmark programs to define reuse library elements for RASSP. More details can be found in the case studies that document these efforts. See the SAR, ETC4ALFS on COTS Processor, and SAIP case studies.

7.3 Additional Requirements

Aside from the VHDL or software code and test benches that define the functionality and performance of each element, non-functional requirements, such as size, weight, power, cost, or reliability numbers are incorporated into the structure of the reuse objects. This provides users with important data about the reuse elements that they require to judiciously select appropriate pieces for reuse. The long-term goal is to provide, in a standardized format, data that can be used by high-level (architecture trade-off and synthesis) tools to support design trade-offs and optimization. By incorporating this full set of functionality and features as part of the object-oriented reuse hierarchy, RASSP promises to realize reuse to a greater extent and with greater efficiency than ever before.

RASSP Model Year Architecture (MYA) Appnote

8.0 Specifications

Under RASSP, the Model Year Architecture has been documented in the following specifications. For more detail beyond this application note, please refer to this specification:

Model Year Architecture Specification, Version 1.0:

- Vol. I -Introduction to Model Year Architecture, Sept. 5, 1996.
- Vol. II - Hardware Architecture Element Specification, Sept. 20, 1996.
- Vol. III - Standard Virtual Interface Specification, Sept. 23, 1996.
- Vol. IV - Standard Virtual Interface Specification Appendices, Nov. 25, 1996.

- Vol. V - Software Architecture Element Specification (Not Available)
- Vol. VI - Model Year Architecture Re-Use Library Element Specification (Not Available)

RASSP Model Year Architecture (MYA) Appnote

9.0 References

9.1 Documents

RASSP Methodology Document, Version 2.0, Volume I, Lockheed Martin Advanced Technology Laboratories, October, 1995. [METHODOLOGY_95]

[PGM] Processing Graph Method Specification: Version 1.0, Navy Standard Signal Processing Program. (PMS-500), December 1987 [To obtain these documents and other information regarding PGM refer to the PGMT home page presented by the Processing System Section, Advanced Information Technology Branch at the Naval Research Laboratory, Washington, D.C. www.ait.nrl.navy.mil/pgmt/pgm2.html or contact Mr. David Laplan at phone number (202) 404-7338 or e-mail kaplar@ait.nrl.navy.mil]

"VSIA Alliance Architecture Document Version 1.0," Virtual Socket Interface Alliance, http://www.vsi.org/library//vsi-or.pdf, 1997.

Standard Virtual Interface
Chhabra, A., "SVI Verification Study: Encapsulations of the Benchmark II-Data I/O Board and the Mercury Rino-RIC Chipset," Lockheed Martin Advanced Technology Laboratories, May 17, 1996. [Data_IO_RINO]

Buchanan, G., "Hardware Synthesis Study of WSSPT SVI Interface Encapsulations," Lockheed Martin Advanced Technology Laboratories, July, 1995. [PCI_SVI]

Reconfigurable Network Interface
Buchanan, G., "RACEway to SVI External Network," Lockheed Martin Advanced Technology Laboratories, July 30, 1996. [RACE_SVI]

Buchanan, G., "Myrinet to SVI External Network Interface," Lockheed Martin Advanced Technology Laboratories, July 31, 1996. [Myri_SVI]

Wedgwood, J., "Reconfigurable Network Interface (RNI) Study: Myrinet to PCI," Lockheed Martin Advanced Technology Laboratories, August, 1997. [Myri_PCI]

Buchanan, G., "SVI Verification Task, Case 2 - RNI Encapsulation Study VHDL Model Description (Simple RNI) -Encapsulation Study Using the Cypress HOTLink high-speed serial link and the PCI fabric interface to implement a sensor-to-PCI RNI," Lockheed Martin Advanced Technology Laboratories, Sept. 20, 1995. [Hotlink_PCI]

9.2 Papers

Wedgwood, J., and G. Buchanan, "A Model-Year Architecture Approach to Hardware Reuse in Digital Signal Processor System Design," Proceedings High-Level Electronic System Design Conference, San Jose, CA, October, 1997, pp 493 - 504. [Wedgwood_97]

9.3 Application Notes

Methodology
Hardware/Software Codesign
Design for Test
Data Flow Graph Design
Terminology and Taxonomy
Token-Based Performance Modeling
Autocoding for DSP Control

9.4 Case Studies

Synhetic Aperture Radar [SAR-CS]
ETC4ALFS on COTS Processor [UYS-CS]
Semi-Automated Image intelligence Processor [SAIP-CS]

Next Up Previous Contents
Next: Up: Appnotes Index **Previous:**8.0 Specifications