Rapid Prototyping of Application Specific Signal Processors

RASSP

Process - Architecture - Framework

Rapid Prototyping of Application–Specific Signal Processors (RASSP)

RASSP Model Year Architecture Specification Volume II

Hardware Architecture Element Specification

Version 1.0

September 20, 1996

ADVANCED TECHNOLOGY LABORATORIES

LOCKHEED MARTIN

Table of Contents

1. INTRODUCTION 1

1.1. Purpose 1

1.2. Scope 1

1.3. Document Outline 2

2. FUNCTIONAL ARCHITECTURE 2

3. INTERFACE APPROACH 5

3.1. Standard Virtual Interface 5

3.2. Internal Module Interfaces 8

3.3. External Interfaces 9

3.4. System Implications of Layering 11

3.5. Application Guidelines 11

3.5.1. Partitioning of Re–Use Elements 11

3.5.2. Selecting Signals for Encapsulation 12

3.5.3. Encapsulation of Commercial–Off–The–Shelf (COTS) Controllers 12

4. TEST ARCHITECTURE 13

5. RE–USE LIBRARY COMPONENT DEVELOPMENT 13

5.1. System Specification View 16

5.2. System Architecture Performance View 16

5.3. System Architecture Behavior View 16

5.4. System Module Interface View 17

5.5. System Structure View 17

5.6. Module Performance View 17
5.7. Module Behavior View 17
5.8. Module Component Interface Behavior View 18
5.9. Integrated Circuit Behavior View 18
5.10. Integrated Circuit Function View 18

6. INTERFACE STANDARDS 19

6.1. RASSP Recommended Selection Process 19
6.1.1. Classification of Interface Standards 19
6.1.2. Identify Viable Open Interface Standards 20
6.1.3. Technical Evaluation of Viable Standards 21
6.1.3.1. Technical Evaluation – First Tier 21
6.1.3.2. Technical Evaluation – Second Tier 22
6.1.4. Quantifying the Selection Process 24

7. SUMMARY 27

1. INTRODUCTION

1.1. Purpose

The purpose of this document is to convey a formal specification for hardware architecture elements which comprise the RASSP Model Year Architecture (MYA) Functional Architecture. The Functional Architecture defines the necessary components at the architectural level and the manner in which their interfaces must be defined to ensure that the resulting architecture design is upgradable and facilitates technology insertion. As such, the Functional Architecture is a starting point for developing an architecture for an application–specific problem, *not* a detailed instantiation of an architecture. Adherence to this specification will ensure the creation of architectural elements, and the design of specific architectures which include them, which will provide the RASSP MYA features of architectural element reuse, interoperability, and facilitated upgradability.

1.2. Scope

The scope of the hardware architecture element specification is defined to include the following:

– **Functional Architecture Description and Design Guidelines**

– The RASSP Functional Architecture is described along with a set of design guidelines and constraints for general architectural development, such as how to properly use the functional architecture and general use of encapsulation libraries.

– **Hardware Element Interface Approach and Design Guidelines**

– The Model Year Architecture approach to interfacing various signal processor components in the functional architecture is defined. Two approaches are defined in detail: that for interfacing internal modules in the processor architecture, and that for interfacing external elements to the processor architecture. This approach is primarily responsible for the high degree of upgradability, interoperability, and insertion capability of the Model Year Architecture.

– The standard functional interface employed to carry out the MYA hardware element interface approach, known as the Standard Virtual Interface (SVI) is detailed in a separate set of specifications: Model Year Architecture Specification Volumes III & IV – RASSP Standard Virtual Interface Specification & Appendices. An introduction to SVI is offered here in Volume II.

– **Encapsulated Library Component Implementation Guidelines**

– The functional architecture describes a multi–viewed VHDL modeling hierarchy for defining architectural level reuse library components. Guidelines for generating and implementing each level of the VHDL model hierarchy is presented.

1.3. Document Outline

Section 2 provides a description of the RASSP Functional Architecture, along with design guidelines and constraints for general architectural development. Section 3 describes the MYA architectural level component interface approach. Section 4 discusses the MYA approach to defining the test architecture of a RASSP processor design. Section 5 defines the guidelines for generating and implementing encapsulated reuse library components. Section 6 describes a recommended methodology for selecting from among open interface standards for incorporation into RASSP designs.

2. FUNCTIONAL ARCHITECTURE

The MYA Functional Architecture defines the necessary components at the architectural level and the manner in which their interfaces must be defined to ensure that the resulting architecture design is upgradable and facilitates technology insertion. As such, the functional architecture is a starting point for developing an architecture for an application–specific problem, *not* a detailed instantiation of an architecture. The diversity of application domain requirements precludes specifying instantiation details at this level. Experience has shown the need to significantly tailor architectures for different application domains. While every design should start with the Functional Architecture, specific architectural details needed to serve the application problem at hand, such as what type(s) of and how many processing elements (PEs) to use, the best interconnect topology for the problem domain, the need for shared memories, etc., will be developed as part of the Architectural Selection portion of the RASSP Methodology. Figure 1 illustrates the relationship of the MYA Functional Architecture to the overall Model Year Architecture Framework. Readers are also encouraged to refer to Section 3. of Vol. I of the MYA Specification, for an overview of the RASSP Model Year Architecture. The Model Year Functional Architecture specifies:

– *A high–level starting point from which to launch application–specific architecture selection.*

– *The use of open standards for the interconnect fabric, sensor, and interchassis interfaces.* (Note: “interconnect fabric” is used in this document to refer to any structure that provides the physical paths necessary for internal module communication.*)*

– *A standard approach for implementing interfaces.*

The Model Year Functional Architecture DOES NOT specify:

– *The topology or configuration of the interconnect fabric. This is application–specific and is determined during the architecture selection portion of the RASSP Methodology.*

– *The specific processor types to use.*

– *System–level interfaces (these depend on other platform subsystems that were previously designed). RASSP signal processors must support a large number of existing and emerging interface standards.*

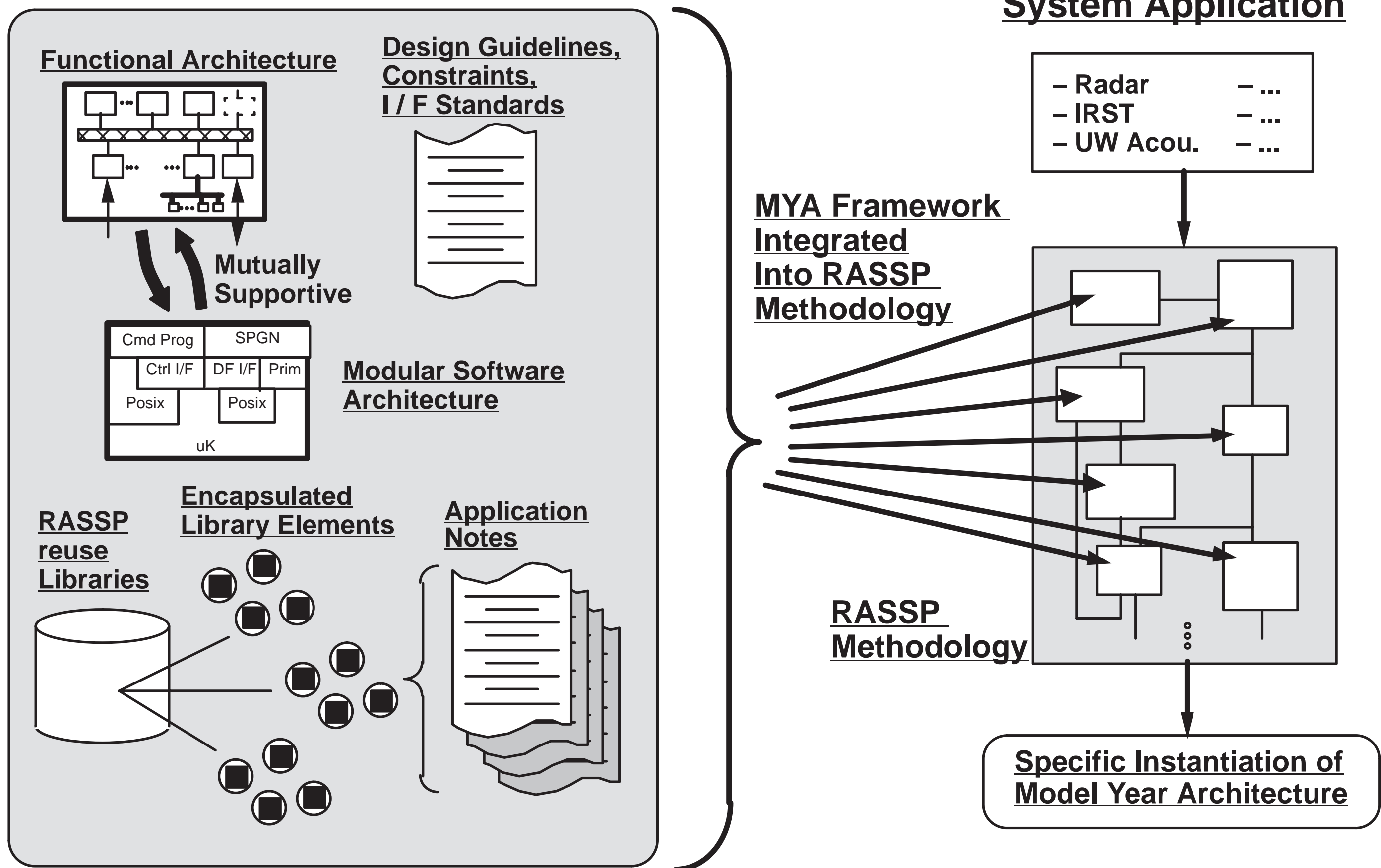


Figure 1 User's View of Model Year Architecture

Figure 2 shows a high–level view of the Model Year Functional Architecture. The depiction is purposely general to serve the needs of various application domains. The only portions of the architecture that are specified are 1) the use of an open standard interconnect fabric, 2) identification of where interface standards should be used, 3) an approach to implementing internal module interfaces, 4) and an approach to implement various external interfaces called a Reconfigurable Network Interface (RNI). The Functional Architecture centers around the interconnect fabric which serves as a backbone for the signal processor. All internal nodes, i.e. processing elements, shared memories, and elements which interface the signal processor to the outside world, communicate through the interconnect fabric through one or more ports. The interconnect fabric is not necessarily only a backplane, as this implies physical limits on its implementation (such as restricting the interconnect fabric to rows of connectors in a cabinet). The interconnect fabric may physically extend onto boards or modules. Additionally, no physical restrictions on the definition of an internal module is implied. This provides flexibility in system partitioning and allows available technology to drive the appropriate partitions.

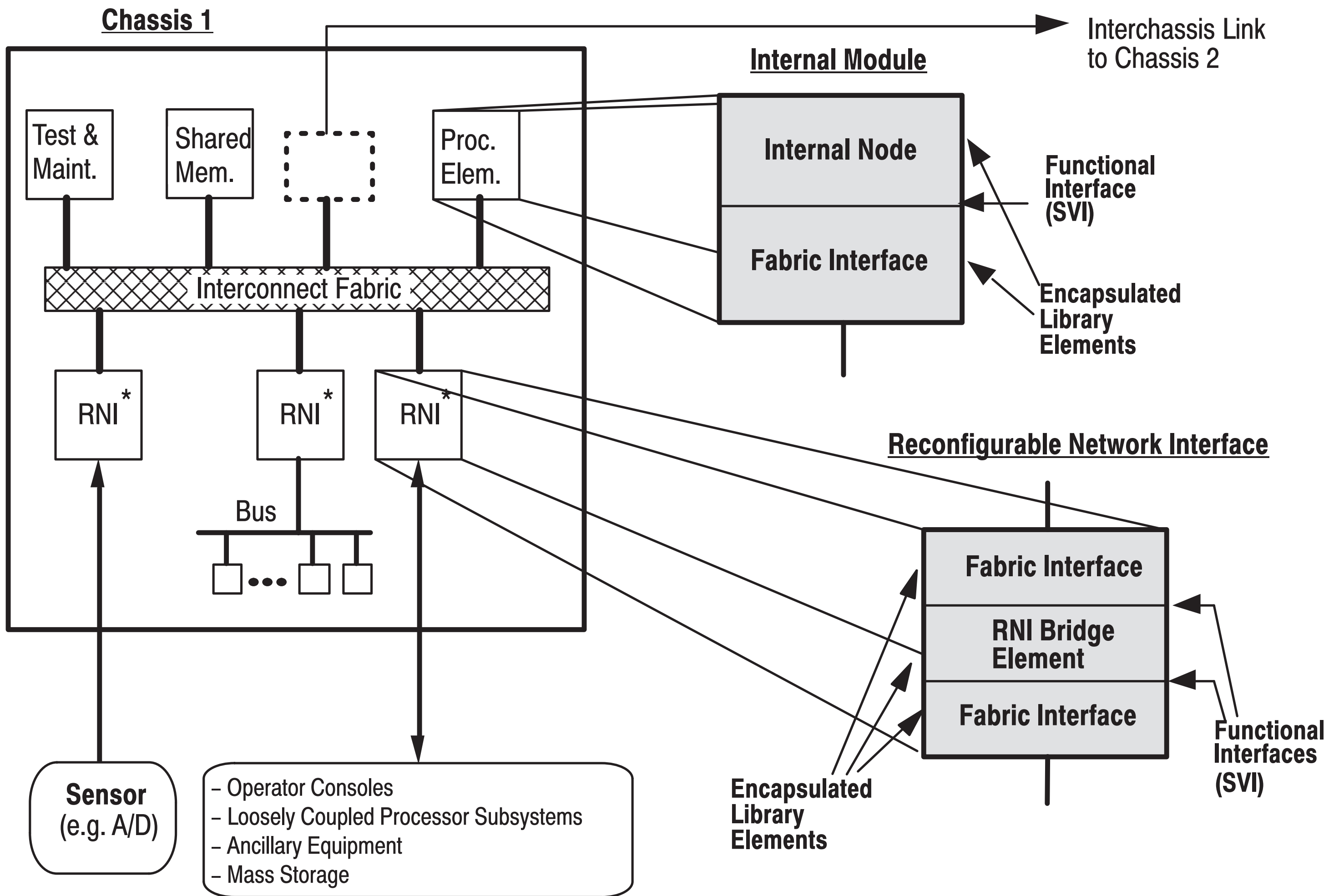


Figure 2 High Level View of Functional Architecture and Interface Definitions

The key aspect of the Functional Architecture is the approach to implementing the various 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 that has been defined is one based on a combination of two concepts: 1) architectural layering, and 2) the use of standard technology–independent functional interfaces (refer to Figure 2). The primary reasons for using a layered approach are that it provides logical decomposition into smaller, more manageable, understandable, reusable, and maintainable parts. Most importantly, it 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. Defining a standard at this level provides the following benefits:

1) *Technology–independence greatly enhances model year upgrade / technology–insertion capability.*

2) *Significant reduction in time required to design new interfaces.*

3) *Interface access to all library elements with same I/F, reducing library maintenance time and cost.*

The use of standard technology–independent functional interfaces at key points within a layered architecture provides a powerful mechanism for achieving the MYA goals of rapid low–cost upgrades and technology insertion. This interoperability is achieved through the definition of a VHDL wrapper for each architectural–level reuse library element that implements a standard functional interface. This wrapper implements the hardware portion of the functional interface referred to as the Standard Virtual Interface (SVI) which is introduced in Section 3.3, and described in detail in Volumes III & IV of the MYA Specification. The wrapper is said to "encapsulate" the library element to hide implementation details, providing only an interface definition (the functional interface) to the user.

Section 3. discusses the approach to defining hardware interfaces throughout the Functional Architecture in more detail, including an introduction to the Standard Virtual Interface.

3. INTERFACE APPROACH

One of the most important aspects of the Model Year Architecture is the approach to interface the various signal processor components. These components include PEs (general–purpose processors, DSPs, special–purpose processors, and hardware accelerators), shared memories, sensors, and subsystem components (ancillary equipment, mass storage devices, etc.). The high importance placed on the interface approach is a result of its impact on the upgradability and technology insertion capability. The following sections describe an approach to defining interfaces to architectural elements that focuses on providing a capability to upgrade systems through technology insertion while simultaneously minimizing and localizing both hardware and software redesign through the concept of layering.

3.1. Standard Virtual Interface

The standard functional interface which has been defined for the RASSP Model Year Architecture is know as the Standard Virtual Interface (SVI). The SVI enables the interoperability and upgradability of architectural level reuse library elements by defining an interface protocol and implementation approach for reuse element encapsulation. Encapsulated library elements (encapsulations) are categorized into three general types of reuse element: Internal Nodes, Fabric Interfaces, and RNI (Reconfigurable Network Interface) Bridge Elements. An internal node is essentially any architectural–level element which becomes connected (through a fabric interface) to the system interconnect fabric. Examples of internal nodes include signal processors, vector processors, or shared memory. A fabric interface is an element which translates between the Standard Virtual Interface of an internal node or RNI element, and the protocol of the interconnect fabric. The term "interconnect fabric" is used to describe any form of node–to–node communication medium. Examples of interconnect fabrics include crossbar–based point–to–point interconnect networks, rings, and

multidrop buses. In Model Year Architecture nomenclature, the joining of an internal node and a fabric interface results in an Internal Module; an architecture–level element as it appears at the system level. Finally, an RNI bridge element is a specialized SVI–to–SVI bridge which sits between two fabric interface encapsulations, resulting in a fabric–to–fabric link which together is called an RNI. Figure 2 illustrates these Model Year Architecture reuse elements, and how they interface with the SVI in the MYA Functional Architecture.

Figure 3 illustrates the concept of the SVI. Each library element, in this case a PE and a COTS interface controller, includes an encapsulation wrapper which implements the SVI. The wrappers are described in VHDL code, and are written in Register Transfer Level (RTL) style which enables the logic described by the code to be synthesized into a targeted PLD, FPGA, or ASIC device. During the synthesis process, the wrappers from adjoining encapsulated elements (PE and fabric interface, for example) are combined to create the PE-to-controller interface device. Note that as an alternative to this example, the COTS controller could be replaced by custom fabric interface logic, also described in VHDL code. In this alternative case, the wrappers as well as the interface logic VHDL would be combined and synthesized into a single interface device, instead of the two devices (COTS and wrappers) shown in Figure 3 .

Not every library element will support every feature of the SVI. For example, an interface that has only one physical or virtual channel into an interconnect network does not require the Channel ID (refer to Volume III), which can therefore be left unused. The SVI is defined such that any internal module may be interfaced to any fabric interface even if only a subset of the functionality is supported for the pairing of two particular elements.

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 synchronous with the SVI clock. Support for asynchronous operation between an internal node and the interconnect fabric can be handled by the SVI encapsulation logic.

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 bidirectional data interface can be obtained by using both unidirectional data interfaces controlled by the SVI master and the SVI slave. The detailed specification of the SVI, with examples, may be found in MYA Specification Volumes III & IV – RASSP Standard Virtual Interface Specification.

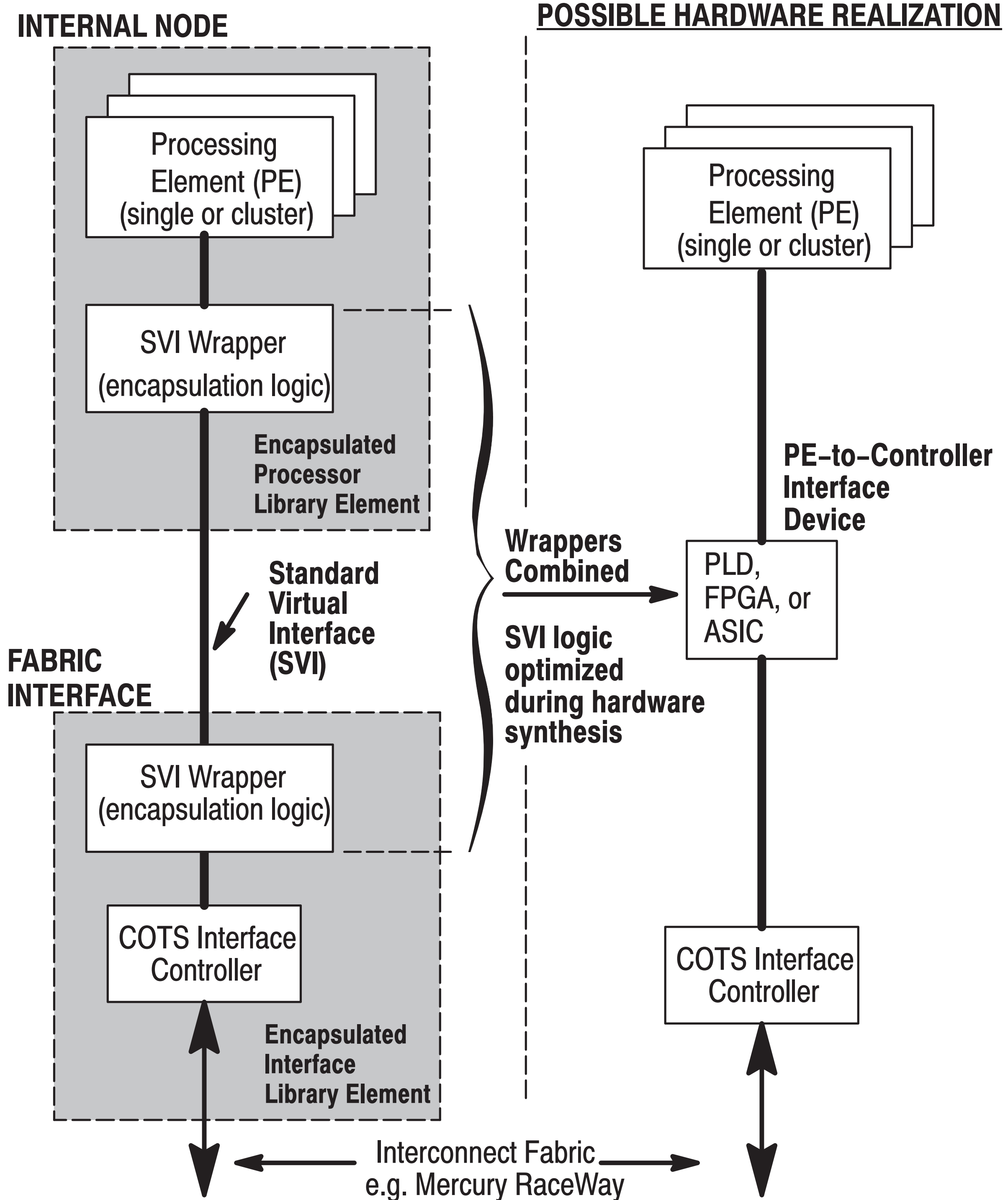


Figure 3 Standard Virtual Interface Approach using Internal Node–to–Fabric Interface example.

3.2. Internal Module Interfaces

Internal modules form the heart of any RASSP signal processor. These are primarily the PEs that implement all the signal processing functions performed in the system. They are given much attention in the design process because their choice and implementation significantly impacts the performance, size, weight, and cost of the signal processing system. For this reason, PEs and their associated interconnect fabric are prime targets for upgradability and technology insertion, which requires an interface approach that supports rapid low–cost hardware and software upgrades.

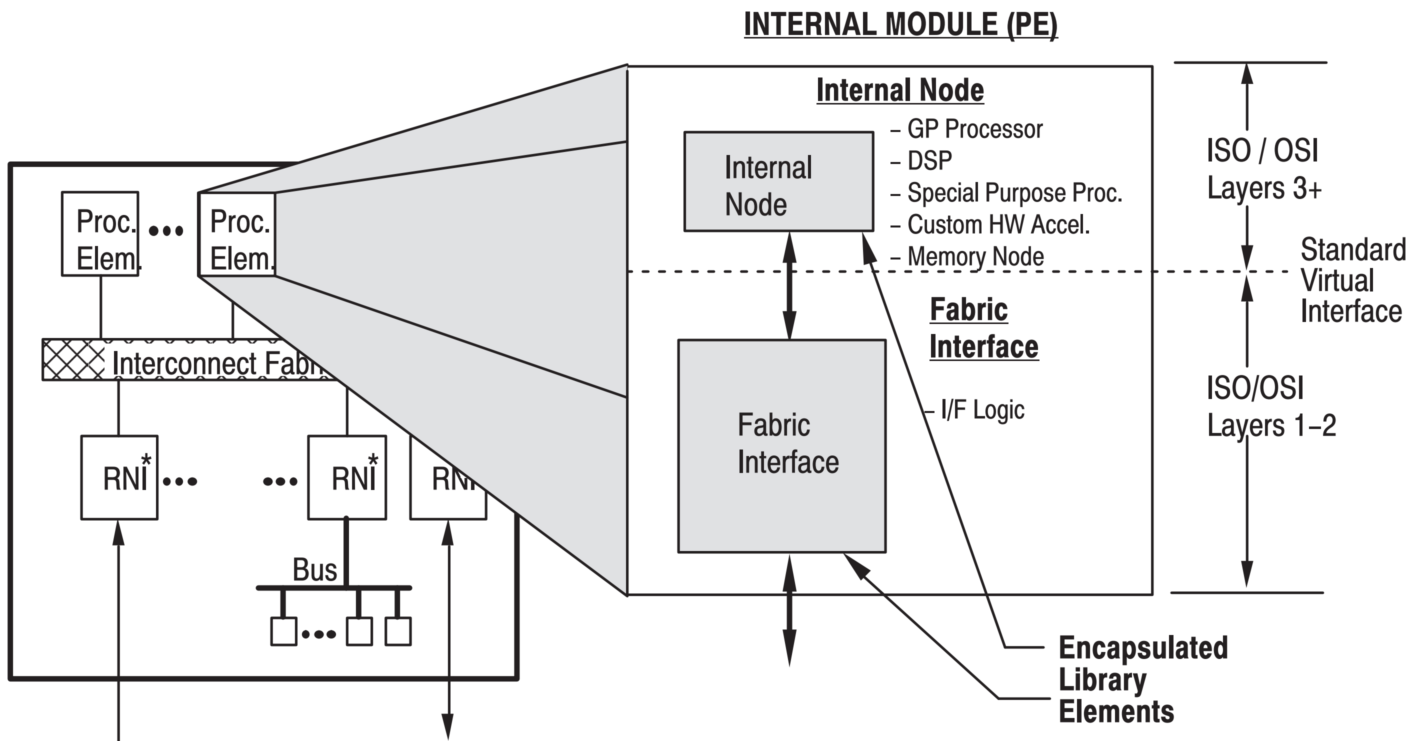


Figure 4 Internal Module Interfaces

Figure 4 illustrates the architectural approach for defining internal module interfaces. The characteristic feature of this approach is the formal separation of the functions of the internal node (PE, memory node, etc.) from the fabric interface. Making the analogy to the ISO/OSI reference model, the lowermost communication layers, that is the physical (1) and the data link (2) layers, are implemented within the Fabric Interface. The Fabric Interface may be constructed of any or all of the following: 1) custom random logic, 2) field–programmable gate array (FPGA), or 3) an application–specific integrated circuit (ASIC). Higher levels, from the network layer (3) and upward, would be implemented within the internal node itself. The Internal Node may be a single PE or a small cluster of PEs, custom hardware accelerators or a shared memory node. Note that 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. The important point is the formal

definition and use of layering to isolate changes and facilitate interoperability of system components when a particular component (Internal Node or Interconnect Fabric) is upgraded.

The key to successful application of a layered approach is defining the layer boundaries to minimize layer interactions and to restrict layer interfacing to only upper and lower adjacent layers, while maintaining a high degree of performance and implementation efficiency. The layer interfaces should generally be standardized. However, to provide the level of interoperability and upgradability for the Model Year Architecture approach to be successful, the interface between the Internal Node and the Fabric Interface MUST be standardized. As previously discussed, the Standard Virtual Interface implements this standard functional interface.

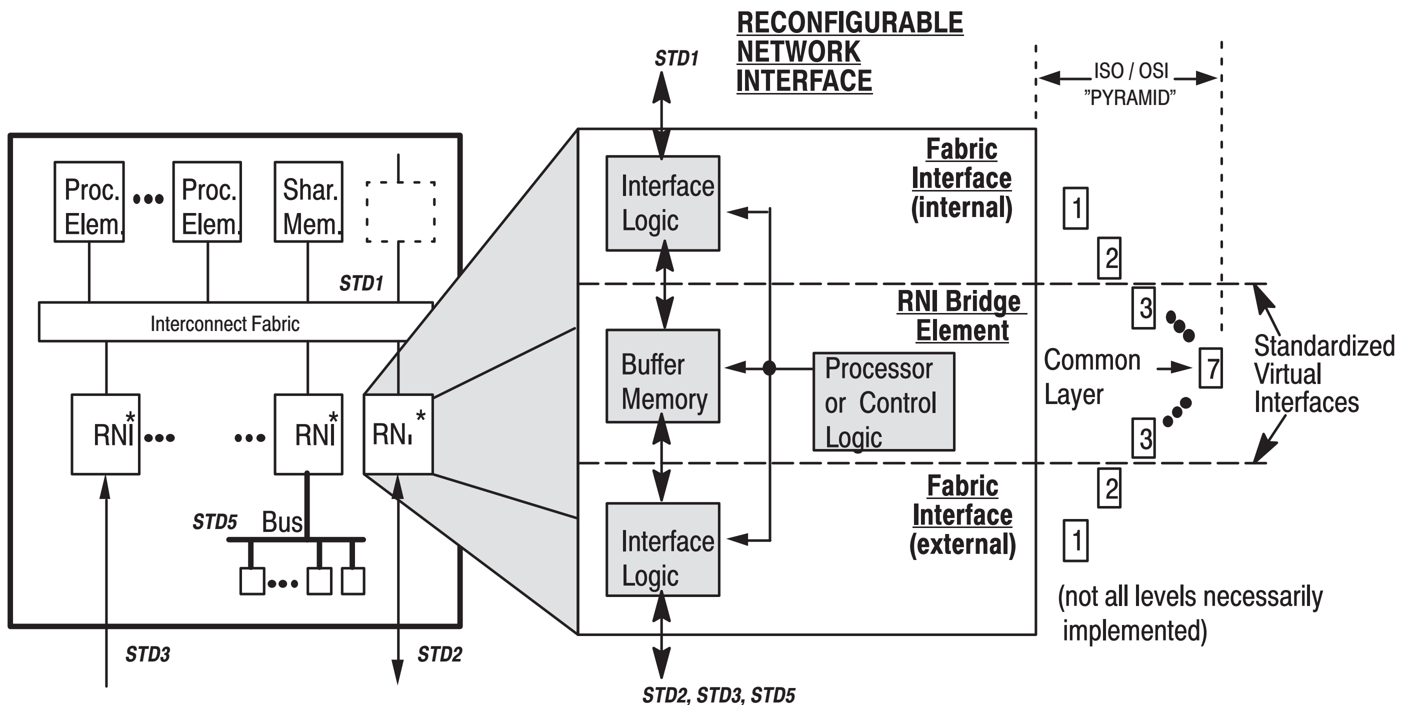
3.3. External Interfaces

An approach must be defined to interface RASSP signal processors to sensors and other platform subsystems. This section describes an architectural approach called a Reconfigurable Network Interface (RNI) for implementing such interfaces. Like the architectural approach for defining internal module interfaces, the concept of formal separation of an internal or controlling node and the network or fabric interface by an SVI is applicable to the RNI. Unlike the internal module interface, which serves to interconnect internal nodes via the interconnect fabric, the RNI is a bridging element between the *internal* interconnect fabric and a particular *external* interconnect fabric. A specific example of this might be a bridge between a fabric based on the Scalable Coherent Interface (SCI) and a MIL–STD–1553B network that interfaces to a navigation subsystem. The RNI architecture is illustrated in Figure 5 .

The RNI is divided into three logical elements: 1) Fabric Interface (internal), 2) RNI Bridge Element, and 3) Fabric Interface (external). The fabric interfaces implement the specific interfaces to the internal interconnect fabric and the specific external interconnect fabric under consideration. 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 interfaces, a layered communication approach is employed. Here, since the RNI provides a bridge between two interfaces, two separate layered structures exist. Shown in Figure 5 , the two layered structures form a pyramid, with the lowest layers of the two interfaces implemented in the Fabric Interface components. The higher levels of each structure are implemented within the RNI bridge element and converge where the data interchange is stripped of all its interface specific identity by the lower layers, effectively performing a protocol conversion. 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.

Two categories of the RNI can be defined based upon the bridge element implementation: the *custom RNI* — employing a custom hardware bridge element implementation — and the *programmable RNI* — employing a microprocessor–based bridge element. In general, either imple-



Fabric Interface (internal) – Implements lowest levels of Interconnect Fabric (e.g. ISO/OSI 1. Physical, 2. Data Link)
RNI Bridge Element – Handles overall control and data buffering of RNI and (optionally) higher level protocols or protocol conversion.
Fabric Interface (external) – Implements lowest levels of External Network (e.g. ISO/OSI 1. Physical, 2. Data Link)

Figure 5 External Interfaces using a Reconfigurable Network Interface

mentation may be chosen for a given application, however the custom RNI will tend toward higher design complexity with lower latency, while the programmable RNI will tend toward lower design complexity (and higher reuse due to programmability) and higher latency. The RASSP system designer must make the necessary performance/complexity trade-offs for each particular application. The greater the complexity of the external network and interconnect fabric which is being bridged, the greater the complexity of custom hardware bridge element design. In the case of bridging between relatively complex protocols, the burden of the bridging function may be more easily implemented in software rather that executing a complicated custom hardware design. In cases where the protocol conversions are simple, or low latency is critical, the custom hardware approach will be more attractive.

The exact implementation of the RNI itself depends on the type of system interface being served. A RNI to implement interfaces for loosely coupled processing subsystems, operator consoles, and certain types of ancillary equipment will have one implementation, while a 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. On the other hand, a local area network application most likely would require a pro-

grammable processor to implement a more complex software–based protocol. More than one class of RNI will be required to support all aspects of signal processor interfacing, and as described in Section 2., would be supported through various object subclasses or derived classes for the Fabric Interface, RNI Bridge Element, and External Network Interface within the reuse library.

3.4. System Implications of Layering

As previously stated, the primary reasons for using a layered approach to hardware architecture design is that it provides a logical decomposition of the system into smaller, more manageable, understandable, reusable, and maintainable parts; and perhaps most importantly, it minimizes and confines changes that are introduced as a result of technology insertion. This latter point is of prime importance to the model year upgrade philosophy. Layering has the potential for incurring some disadvantages. A layered architecture necessarily creates more interfaces and encapsulates functionality within the various levels, restricting how interactions between the various components can occur. These restrictions introduce levels of indirection between component interactions that can decrease efficiency and therefore adversely impact performance. Additional impacts to the system design would include an increase in software code size and an increase in physical hardware required to realize system components.

The advantages to be gained by layering are not automatically reaped by its use. Incorporating excessive layers will introduce unacceptable performance penalties or an unacceptable growth in hardware. However, judicious layering can minimize the performance and size penalties, while offering the advantages of reusability, maintainability, and upgradability. In some cases, tradeoffs must be made between required functionality and acceptable performance. For example, in the design of a RNI for an interface to a local area network, support for higher level protocols would be needed, requiring additional layers to implement them. Some overhead in performance may be acceptable, particularly if hard realtime response is not a consideration. In contrast, a RNI to implement a bus interface would require minimal layering since the protocols involved are at a fairly low level. Performance overhead in this case would not be acceptable. While this overhead can be minimized by careful application of layering, its existence must be accepted as a tradeoff to realize the greater benefits of design and life cycle and cost reduction.

In general, the approach to architecture design in RASSP is to incorporate judicious layering to facilitate the reusability, maintainability, and upgradability required to support the model year philosophy, while allowing for optimization for different applications by trading functionality for performance. Layer definitions are to be incorporated only where necessary to isolate major hardware components from changes which would otherwise be incurred as a result of upgrades and technology insertion.

3.5. Application Guidelines

3.5.1. Partitioning of Reuse Elements

The MYA specification does not dictate the placement of the functional interface (SVI) within the RASSP processor architecture. As such, the selection of the optimum placement of the func-

tional interface, that is, the determination of how system architecture components or groups of components will be partitioned to create reuse library elements, is left to the discretion of the system designer. As discussed in Section 3.4., there is performance and hardware overhead associated with the introduction of additional hardware layers. Ultimately, the introduction of functional interfaces into a processor architecture should be based on the design tradeoffs for the particular system application. In general however, reuse elements should be created on the basis of "architectural–level" components. An architectural–level component is considered a functional unit, referred to as an internal node in the discussions in Section 2., such as a processor or cluster of processors, a shared memory module, or a hardware accelerator board, *without* its fabric interface (which is a separate reuse element). In general, the partition of an architectural–level component should also coincide with the definition of LRMs (Line Replaceable Modules) for a particular application. The introduction of the functional interface at lower levels of functionality will introduce additional performance penalties without providing benefits in terms of upgradability or reuse.

3.5.2. Selecting Signals for Encapsulation

The functional interface concept in general, and the Standard Virtual Interface specifically, is designed to facilitate the interoperability of architectural–level components *operating on disparate data communication protocols*. SVI accomplishes this interoperability by defining a FIFO–like–interface message passing protocol which provides handshaking to indicate: 1) when a source would like to initiate a message transaction to a target, 2) when the target is ready to receive data from the source, and 3) when the source is sending valid data. Any communications protocols at ISO/OSI levels 1–2 essentially communicate this same type of information: the *data* to be transferred and *synchronization* signals from the sender and receiver. The functional interface is *not* appropriate for communicating a random collection of synchronous or asynchronous signals at some arbitrary slice through some system logic. Although it is completely possible to force any combination of signals into a representation on the functional interface, the design will likely be complicated, will introduce high latencies due to interrupting the normal data stream to communicate changes in other signal states, and the resulting meaning of the data transfers across the functional interface will be so unique to the specific encapsulation, that true interoperability will be prevented without extensive rework preceding interconnection with another reuse components.

3.5.3. Encapsulation of Commercial–Off–The–Shelf (COTS) Controllers

As discussed in MYA Specification Vol. III & IV, fabric interface encapsulations can be accomplished by encapsulating COTS controllers of the desired interconnect fabric. One caution needs to be recognized however. A bus controller–type device often contains much more functionality than simply a protocol engine for the interconnect fabric. If the controller is meant as an interface to a processing node or cluster, it may contain such additional processor–support functions as DMA engines, system performance monitoring, and error detection and correction. This additional functionality is accompanied by associated data and control signals in addition to those specifically related to the interconnect fabric data. As discussed in Section 3.5.2. above, the introduction of signals outside the purview of those that can be classified as "interconnect fabric signals" into the functional interface will likely result in an encapsulation that is difficult to implement, and whose performance is highly degraded. Although possible, it is generally not advisable to attempt to encapsulate a fabric

interface by encapsulating such a function–rich interface controller. Most likely, these functions — which were placed there for the benefit of a specific processor technology — will have no use when taken outside the context of their native processors, and joined with the fabric interface.

4. TEST ARCHITECTURE

As described in MYA Specification Vol. I, RASSP signal processors must incorporate a test strategy as an integral part of the design to ensure a high–quality product while maintaining the aggressive goals of design cycle, non–recurring and life cycle cost reduction. The support of this test strategy however, does not directly impact the RASSP Model Year Architecture beyond that of requiring that the test architecture incorporate standard test interfaces that will become required augmentations to normal signal and control interfaces to chips, modules, and systems.

Given that the test interface incorporates an open test standard, most likely IEEE 1149.1 or IEEE 1149.5, it would be of no value to add an additional hardware layer on top of this standard interface. As such, the RASSP functional interface (SVI) is not involved in interconnecting standard test structures. If an application has need of a non–standard test interface, it can be transported on the defined SVI signals, without a need to define dedicated test channels.

For further information on the RASSP test methodology refer to *RASSP Design For Testability (DFT) Methodology.*

5. REUSE LIBRARY COMPONENT DEVELOPMENT

Although the RASSP Functional Architecture defines architectural level components and their interface definitions, the embodiment of a Model Year Architecture lies in the reuse libraries which support it. The model reuse library consists of a structure of VHDL models which are intended to address both the necessary *level of representational abstraction* and the necessary *level of physical hierarchy* associated with each particular aspect of the RASSP system design process. There exits a wide variety of model types to cover the full range of design activity requirements, from architectural trade–off analysis to detailed integrated circuit design. Succinct definitions of these model types can be found in the *RASSP VHDL Modeling Terminology and Taxonomy.* In the course of the overall development of a RASSP system, any combination of types of VHDL models may be developed, and therefore be placed in the reuse library. Which particular models are developed for any given system will depend on the components being developed, whether or not they are procured "off the shelf," and the particular deliverables that are required in the development contract. Refer to *RASSP Product Data Packages Analysis* for a discussion of recommended modeling requirements for a RASSP system. It is possible however, for illustrative purposes, to describe a typical model structure for a hypothetical RASSP system.

In this typical model structure, we define ten major views of a RASSP system which could be required by the system design process. We refer to the combined levels of representational abstraction and physical hierarchy as the "view" of the system offered by the model. Associated with each view is a "model type" which refers to the model's classification in terms of the *RASSP VHDL*

Modeling Terminology and Taxonomy. Note that while the model that describes a particular view is a reuse component, the *components* of the model are also reuse elements, so the complete model structure of the system may be more complex than these ten views.

View 1.: ***System Specification*** **Model Type:** ***Executable Specification***
A behavioral description of the RASSP system, provided by the customer, which describes the required performance of the system to be designed.

View 2.: ***System Architecture Performance*** **Model Type:** ***Token–Based Performance***
Timing–only behavior to support high level architectural tradeoffs; internal modules (see Figure 2) are leaf–level entities.

View 3.: ***System Architecture Behavior*** **Model Type:** ***Abstract Behavioral***
Full timing and function of multi–component electronic modules, with *abstract* interface descriptions, used for module–level trade–off analysis; internal modules are leaf–level entities.

View 4.: ***System Module Interface Behavior*** **Model Type:** ***Interface***
Timing and function of multi–component electronic modules' *I/O interfaces only,* used to design and verify module interface behavior; internal modules are leaf–level entities.

View 5.: ***System Structure*** **Model Type:** ***Pure Structural***
A hierarchical view of the system structure only, for system–level and module–level interconnect verification and test design; integrated circuits are leaf–level entities.

View 6.: ***Module Performance*** **Model Type:** ***Token–Based Performance***
Timing–only behavior of multi–component modules, for building the System Architecture Performance view, no leaf–level entities.

View 7.: ***Module Behavior*** **Model Type:** ***Detailed Behavioral***
Full timing and function of individual integrated circuits for module–level detailed design and verification; integrated circuits are leaf level entities.

View 8.: ***Module Component Interface Behavior*** **Model Type:** ***Interface***
Timing and function of individual integrated circuits' *I/O interfaces only,* used to design and verify integrated circuit interface behavior; integrated circuits are leaf–level entities.

View 9.: ***Integrated Circuit Behavior*** **Model Type:** ***Detailed Behavioral***
Full timing and function of individual integrated circuits, used for building the Module Behavior view; no leaf–level entities, unless COTS soft cores are used.

View 10.: ***Integrated Circuit Function*** **Model Type:** ***Register Transfer Lvl. (RTL)***
Technology–independent behavior of individual custom (non–COTS) integrated circuits used for circuit synthesis; no leaf–level entities, unless COTS soft cores are used.

Which of these particular model views are used to represent a particular system component depends essentially on where the COTS boundaries lie within the system. The RASSP reuse methodology requires that there is, at a minimum, a detailed–behavioral model for every leaf–level entity in a RASSP system. This requirement assures that any future technology upgrade will at least have a VHDL–executable behavioral specification of every component from which to re–design or re–acquire that component in currently available model year technology. If, for instance, a system con-

tains a sub–system component such as a COTS processor in a chassis, then this chassis is a leaf–level component in this system context, and a detailed–behavioral model of the processor would be required for the reuse library. No models would be required of any components contained within the chassis (boards, modules, or circuits) assuming that these are all part of the COTS component. Alternatively, if a RASSP system contains a COTS integrated circuit, the reuse library would be required to contain a detailed–behavioral model of the integrated circuit, but no RTL model would be necessary.

The RASSP reuse library structure is key to supporting the RASSP design methodology from two important standpoints: it facilitates a seamless top–down design approach, as described in detail in the *RASSP Methodology,* and it facilitates a reuse strategy that is not constrained by the implementation technology targeted at the time of library component's creation. The RASSP design process entails proceeding from a highly abstract virtual prototype – termed VP0 – describing the key functional and performance features of the system, down to a highly detailed virtual prototype – VP3 – which contains detailed hardware and software design implementation details. One advantage of the top–down virtual prototyping approach is that each stage of the virtual prototype is essentially a copy of the more abstract virtual prototype above it, but containing more detailed, less abstract, instantiations of the component descriptions. For example, a VP2 virtual prototype used for architecture verification would be comprised of View 3 models; leaf–level cells of modules (boards or MCMs) described as behavioral entities with timing information at the module I/O level. As integrated circuit–level descriptions (View 9) of the components which comprise the modules are modeled and instantiated into VP2, it becomes a VP3 – a low–level detailed description of the system. Moreover, a top–down virtual prototyping design methodology, supported with the reuse model hierarchy, allows the same basic virtual prototype structure to be matched to the granularity of the problem under analysis, from architecture selection at one end to detailed circuit design on the other.

The RASSP reuse model structure also ensures that reuse elements will have a useful life even as technological advances render the lower level, technology–dependent, implementation models obsolete. Assume that a complete reuse model structure has been created for a system, including: a token–based performance model, where each of the internal modules are partitioned onto single boards; abstract behavioral models of each of the boards; and RTL models of each of the custom integrated circuits on each board. All these models would then exist in the reuse library for future use. One model year upgrade may find that the integrated circuit technology has advanced in such a way so as to enable a major reduction in clock cycle time. In this case, the model year upgrade may necessitate a complete change to the RTL models of the integrated circuits, but the higher level models could all be reused as is, since all the higher–level functionality would remain unchanged. Another model year upgrade may be accompanied by advances in both integrated circuit technology and packaging technology – enabling the repartitioning of the system with entire boards packaged onto single MCMs, and multiple PEs onto single boards. This model year upgrade would necessitate changes to both the RTL (and detailed–behavioral) models of the integrated circuits and the abstract behavioral models of the boards, since the MCM implementation caused changes to board–level partitioning. The token–based performance model could still be reused however, since the functionality at this level would remain unchanged. Finally, if a model year upgrade includes all the above changes in addition to changes to the system architecture, at this point only the original executable specification would remain as a useful reuse element. The power of describing architectural compo-

nents in this hierarchy of abstraction levels from a reuse standpoint, is that as technology changes, it is possible to traverse to higher and higher levels of the abstraction hierarchy, and still reuse important aspects of the models.

The following sections further describe the basic RASSP VHDL modeling structure outlined above.

5.1. System Specification View – Executable Specification Model

The executable specification is a behavioral description of the proposed RASSP system. It is provided by the customer, and the model describes the function and timing of the intended design as seen from the system's interfaces. The executable specification may describe the electrical, behavioral, or physical aspects of the intended design, including power, cost, size, fit, and weight.

5.2. System Architecture Performance View – Token-Based Performance Model

The system architecture performance view is obtained with a token–based performance model. This view is used to perform architecture tradeoffs and to determine details such as the number of processing elements required, the required processor interconnect network bandwidth, memory architecture, and I/O bandwidth. A performance model is a model that measures the ability of a design to process input stimuli in a required period of time, generally in clock cycles. A token–based performance model measures performance at the highest level of abstraction, by modeling the time required to perform functions within a design, and then passing symbolic data "tokens" between the performance model elements. At this level of abstraction, there is no indication of how functions are performed within each performance model element, and no actual data is (or can be) processed by them. The signals between elements in a token–based performance model do not correspond to any physical pins or ports of the actual element implementation, but only represent their functional ports. The details of SVI encapsulation are invisible at this modeling level, but the timing models will include any performance overhead for those elements that contain SVI encapsulations.

5.3. System Architecture Behavior View – Abstract Behavioral Models

The system architecture behavior view is obtained with an abstract behavioral model of the system architecture. The elements of this model describe the function and timing of architectural–level components, without describing the details of their implementation. Unlike a token–based performance model, an abstract behavioral model of a system actually models the correct processing of data presented at the inputs of its elements, and these elements will output their processed data in a timing–correct manner. Also unlike a token–based performance model, the interfaces of the elements of an abstract behavioral model correspond to abstract complex data types representing actual data, instead of tokens. These abstract data types could represent a combination of data and address, for example, but don't correspond to actual physical I/O pins.

The system architecture behavior view is used during architecture verification and detailed design, to perform verification of system software, and to investigate the impact of replacing archi-

tectural elements during a model year upgrade. The details of SVI encapsulation are still invisible at this level of the model, but here again the abstract behavioral model must include any delay penalties introduced by the SVI layer, if one is present.

5.4. System Module Interface Behavior View – Interface Models

The system module interface behavior view is obtained by assembling interface models of the system architecture components. The elements of this model describe the function and timing of architectural–level component's interfaces, without describing the details of the interface implementation or any internal functionality. Data passed between elements is not processed by the elements, but is just "dummy data" used in order to model interface bus cycles and protocols. The model is used to verify inter–element interconnect and the interface circuitry of interconnected elements. The inclusion of SVI encapsulations need not be reflected in the architecture component interface models, as their interface behavior is unaffected by the SVI layer.

5.5. System Structure View – Pure Structural Model

The system structure view actually represents a hierarchy of structural models that convey all structural information about the system, starting from the interconnection of the lowest leaf–level COTS entities, on up to the system level. The lowest level models will generally be multi–component modules or boards (collections of integrated circuits), and higher–level models will correspond to the physical partitioning of collections of these modules into boards, chassis, and sub–systems. Moving down the physical hierarchy, structural modeling will cease whenever a COTS boundary is reached, whether this is an integrated circuit device, a board, or a chassis. The pure structural models are used to document physical interconnect for manufacturing, for interconnect verification during detailed design, and for the creation of structural tests. The presence of SVI encapsulation will not be reflected in the system structure view.

5.6. Module Performance View – Token-Based Performance Model

The module performance view is represented by a token–based performance model of a particular multi–component module (Section 5.2.). The models provided by this view are used as leaf cells in the construction of the system architecture performance view. The models of the module performance view transmit and receive symbolic data tokens based on the function or operation they are asked to perform. The models contain no indication of how the element's functions are performed, and they are unable to actually process real data. The details of SVI encapsulation are invisible at the level of a token–based performance model, but the model must take into account any delay penalty introduced by encapsulated architectural elements. This delay penalty will take the form of the number of addition cycles of latency the SVI layer imposes.

5.7. Module Behavior View – Detailed Behavioral Model

The module behavior view models the behavior of multi–component modules by assembling detailed behavioral models of the module's integrated circuits. These circuit models describe the

function and timing of the integrated circuit components, but do not necessarily describe the specific implementation details of the devices. The models' interfaces correspond to the physical pins of the actual integrated circuit. The module behavior view is used for detailed design, and to investigate the impact of replacing an individual integrated circuit during a model year upgrade. Any delay penalty associated with the SVI encapsulation must be be included in the timing behavior of affected integrated circuit behavioral model.

5.8. Module Component Interface Behavior View – Interface Model

The module component interface behavior view is obtained by assembling interface models of the integrated circuit components. The component interface models do NOT contain the details of internal behavior, but will properly model the interface behavior and timing of the integrated circuit device. As with the module behavior view, in this view input data is not actually processed and any output data is just "dummy" data; however all input and output control signals are fully functional in order to model interface bus cycles and protocols. This view is used to verify intra–module integrated circuit interconnect, and to verify the interface logic of interconnected devices. The inclusion of SVI encapsulations need not be reflected in the integrated circuit interface models, as their interface behaviour is unaffected by the SVI layer.

5.9. Integrated Circuit Behavior View – Detailed Behavioral Model

The integrated circuit behavioral view is provided by means of a detailed behavioral model of individual integrated circuit devices. This model describes the timing and function of an integrated circuit, without implying a specific internal implementation, and describes the function and timing of each individual I/O pin on the device. The integrated circuit detailed behavioral model is used to build the module behavior view, for the detailed design of multi–component modules and for analyzing the impact of model year upgrades of individual integrated circuits. Any delay penalty associated with the SVI encapsulation must be be included in the timing behavior of affected integrated circuit behavioral models.

5.10. Integrated Circuit Function View – Register Transfer Level (RTL) Model

The integrated circuit view uses RTL models to describe the behavior and implementation details of custom (non–COTS) integrated circuits in the RASSP system. An RTL model describes an integrated circuit in terms of registers, combinational circuitry, low–level buses, and control circuits. This description is the lowest level of description of a reuse library element. The integrated circuit RTL model is used for detailed logic design, and ultimately is the source code for directly synthesizing an ASIC, FPGA, or PLD implementation. On those devices that form the interface between the architectural–level elements and the interconnect fabric, generally one per module or architectural element, the SVI encapsulation will also be represented by RTL description. Any delay penalty associated with the SVI encapsulation will automatically be included in the model by virtue of the inclusion of the SVI description itself in the model.

6. INTERFACE STANDARDS

Open interface standards should be used within RASSP systems wherever possible to further ensure interoperability between components. Using commercially accepted and non–proprietary standards may preclude the necessity of functional interface encapsulations is some cases, as COTS fabric interface components may allow direct communication between architectural elements and the selected standard interface. In any case, confining designs to open interface standards will help ensure multiple sources and reasonable costs for compatible components, eliminating the dependence on sole source proprietary components, and will help 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 has been determined 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 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 will be used to interface RASSP signal processors to a large amount of ancillary equipment and subsystems previously designed and/or outside the control of RASSP.

6.1. RASSP Recommended Selection Process

The following process may be used to systematically select the optimum interconnect approach for a particular design from among open interface standards.

6.1.1. Classification of Interface Standards

Interface standards are categorized for convenience in the RASSP selection process into six applications areas, and are identified in Figure 5 .

STD1 – Internal Module Interface

The internal module interface forms the backbone for RASSP signal processors and is therefore the single most important interface in RASSP. Examples of this application area include: SCI, Fibre Channel, Raceway, QuickRing, and SkyChannel.

STD2 – Subsystem Interface

Examples of subsystem interface types include: Ethernet, MIL STD 1553, Heterogeneous Interconnect (IEEE P1335), Fiber Channel, ATM / Sonet, FDD, and IEEE Firewire.

STD3 – Sensor Interface

Three possible general application areas for sensors have been identified: 1) integrated sensors: e.g. multichannel A/D converter modules within the signal processor backplane with analog inputs provided directly into the signal processor cabinet, 2) local sensors: external to the signal processor but located within 1 meter of the signal processor, or 3) remote sensors: 1 – 100 meters or more. In the case of integrated sensors, the sensor interface would simply be the internal module interface. Local sensors may use the internal module interface if it is capable of traversing small distances beyond the backplane itself (possibly with suitable buffering / retiming). Remote sensors will require a specific interface developed specifically for long distance links, examples of which include: Fiber Channel, ATM / Sonet, and HIPPI / Serial HIPPI.

STD4 – Inter–Chassis Interface

High–end systems may require more than one chassis or cabinet to accommodate the signal processor, requiring the capability of high–speed low–latency communication between / among chassis. Depending on the capabilities of the internal module interconnect, inter–chassis communication might be accomplished by its extension between chassis (possibly with suitable buffering / retiming), or a special link may be required, particularly if the physical separation between chassis is more than some nominal distance. Examples of inter–chassis interfaces include: Extension of internal module interface (if suitable), Fiber Channel, and Mercury RLNK.

STD5 – Multidrop Bus

Multidrop busses may serve three purposes in a RASSP system: 1) internal module interconnect for some classes of systems; 2) a means of general control, configuration, or bootstrapping the system; or 3) a means to interface previously designed modules or systems which are based on a particular multi–drop bus. Examples of the more commonly used multidrop buses are: Futurebus+ , Pi–Bus, VME, Multibus II, SkyChannel, and PCI.

STD6 – Test / Maintenance Interface

The use of standard test interfaces throughout the design hierarchy is necessary to support the RASSP Design for Test strategy. Two main application areas for test interfaces are: 1) chip and MCM level, and 2) board, chassis, and system level. The following open test interface standards support these test interface applications: Chip and MCM level: IEEE 1149.1 (JTAG); Board, Chassis, and System level: IEEE 1149.5 Test Maintenance (MTM) Bus, VME Extensions for Instrumentation (VXIbus), and Hierarchical 1149.1 Extensions.

The selection of appropriate test interfaces will be performed with the DFT methodology task, which is also developing the test architecture.

6.1.2. Identify Viable Open Interface Standards

The second task for selecting an optimum open interface standard for a design is to identify the viable open interface standards from among the population of possibilities. The following pro-

grammatic criteria should be considered to identify and evaluate current and emerging open commercially–accepted interface standards as applicable candidates for RASSP signal processors:

- **Availability of Commercial Components**

Components to implement a standard should be commercially available in the time–frame of the particular RASSP model year under consideration.

- **Commercial Acceptance**

The standard should have or show great promise to have wide commercial acceptance.

- **Sufficient Documentation for Design**

The specification must be well documented to alleviate problems associated with ambiguous interfaces.

- **Economics/Cost**

Implementing networks conforming to the standard specification should be achievable by vendors from an economic viewpoint.

- **Maturity**

The standard should be mature with high confidence of implementation using various technologies in the timeframe of the particular RASSP model year under consideration.

- **Non–proprietary and supported**

The network specification / interface must be in the public domain and should not be proprietary.

6.1.3. Technical Evaluation of Viable Open Interface Standards

For a specific standard to make its way into a signal processor design, it should be subjected to a more formal selection process than the initial identification of viable open interface standards, and should addresses the requirements for a particular application domain. Once selected, RASSP encapsulated interface library components would be generated (if none are available from previous designs) to support the selected interface standard. As new standards emerge and are adopted for use, they can be incorporated into new library components to facilitate their interoperability with previously defined components, thus providing a model year upgrade with respect to the new interface. A proposed formal selection process suitable for application specific standard selection is presented in this section.

6.1.3.1. Technical Evaluation of Viable Open Interface Standards – First Tier

Once a set of viable open interface standards has been compiled, the next step is to begin the technical evaluation of these standards relative to both the general requirements of a RASSP archi-

tecture, and the specific requirements of the targeted design. The "first tier" criteria are the most basic technical criteria which will ensure that a particular open interface standard will qualify for consideration in a RASSP design. The applicability of these criteria, and the following second tier criteria, is dependent to some degree on the interface type under evaluation (STD1 – STD6); a matrix of the technical evaluation criteria to be used for each of the various RASSP interface types is presented in Table 1 .

- **Bandwidth**

Minimum usable bandwidth (bits per second) per link required to support high–performance systems developed under RASSP.

- **Physical Constraints**

Specifies the size, weight, pin count, power, etc. that the interface to the network requires.

- **RASSP Compatibility**

The ability of the network to be used in the global RASSP framework with other RASSP standards.

- **Real–time Support**

The ability of the network to support realtime transactions, that is, to perform a transaction within a set deadline.

- **Scalability**

The ability of the network to accommodate from a very small amount of resources (nodes) to a very large amount of resources.

6.1.3.2. Technical Evaluation of Viable Open Interface Standards – Second Tier

Finally, after candidate open interface standards have passed the first tier technical criteria, the remaining standards should be evaluated according to the following addition technical criteria. These additional technical criteria are more appropriately addressed with respect to the requirements of particular applications where the relative importance for each factor can be weighed.

- **Asynchronous Clocking**

The ability of the network to operate without having to distribute a global clock throughout the network.

- **Cache Coherence**

The ability of the network to provide intrinsic support for cache coherency in a distributed shared memory environment.

- **Concurrent Path Scalability**

The ability to connect multiple interconnects between any two nodes to support either redundancy or higher communication performance.

- **Data Security**

The ability to prevent unauthorized access to data in a system.

- **Environmental**

Specifies requirements for shock, vibration, temperature range, radiation hardening, EMI, EMP, EMC, etc. that the network interface must meet.

- **Error Correction**

Specifies the types of errors on the network which are corrected and any extra actions which must be taken to correct errors.

- **Error Detection**

The ability to detect transmission errors and the probabilities that certain types of errors are detected.

- **Fault Tolerance**

The ability of the network to continue to operate after a hard fault (e.g. a stuck at fault on a signal line or a dead or crashed processor node).

- **Flow Control**

The ability of the network to allow a destination node to control (throttle) the speed at which information is sent from the source node.

- **Interconnect Distance**

The physical point–to–point distance over which a network is a able to support communications.

- **Latency**

Maximum time delay of a message from a source node, through the interconnect network, and to the destination node. Latency is a combination of: 1) time of access to the interconnect and 2) transit across the interconnect.

- **Memory Address Space**

The quantity of memory that is directly addressable across the network as well as the type of address space (flat or hierarchical) in a network that supports a memory–based communication paradigm.

- **Message Passing**

The ability of the network to function in a message–passing environment.

- **Message Size**

Maximum and minimum packet sizes supported by the network.

• **Multicast**

The ability for intrinsic support of multicast messaging.

• **Node Addresses**

Number of unique addresses for nodes available in the network.

• **Reconfigurability**

The ability and speed at which the network is capable of activating redundant networks or nodes to compensate for faults or changes (includes live insertion).

• **Routing**

Specifies the routing characteristics supported by the network, i.e. connection versus connectionless, deterministic versus non–deterministic, ability to circumvent deadlocks, virtual channel support, etc.

• **Shared Memory**

The ability of the network to function in a distributed shared memory environment.

• **Software Impact**

The protocol is considered to have no software impact if the software writer does not need to know the exact system configuration to write programs, but only a global address or other similar means of identification.

• **Time Synchronization**

The ability to have a common time base across the network.

• **Topology Support**

The ability of the network to support various topologies.

• **Testability Support**

The ability of the network to support various testability schemes.

6.1.3.3. Quantifying the Selection Process

Figure 6 outlines a suggested process for selecting from among open interface standards for a particular design. This process was adapted from that used by the NGCR working group. The NGCR approach is very comprehensive and provides a good basis for a RASSP selection process. The aim of NGCR is to select a set of standards representing the most widely–accepted, commercially–based standards for an extended period of time. NGCR's nine interface application areas include backplane, point–to–point processor and sensor interconnects, high–performance networks, and local area networks. The aim of RASSP is to provide more flexibility in choice (within a set of suitable

candidates) and is less oriented towards long term use of specific standards, but more toward upgradability and interoperability of those standards. To this end, RASSP simplifies the NGCR selection process by eliminating the formalism of requiring trial runs, specific meetings, and presentations.

Referring to Figure 6 , the list of candidates for a particular interface standard application within a RASSP signal processor is one of two inputs to the overall selection process. The other input to the process is the set of evaluation factors that are assigned weights according to the requirements of the particular application or application domain. The weights are developed by a team prior to candidate evaluation, and are then averaged across the team to provide a final set of weights. The evaluation factors are used to score the interface candidates, the results of which are screened for mandatory requirements. Any candidate not meeting critical evaluation factor(s) is eliminated. The remaining candidate scores are weighted and averaged to form the final scores. The team selects the candidate with the highest score, unless there are obvious scoring anomalies, in which case the selection is made by consensus.

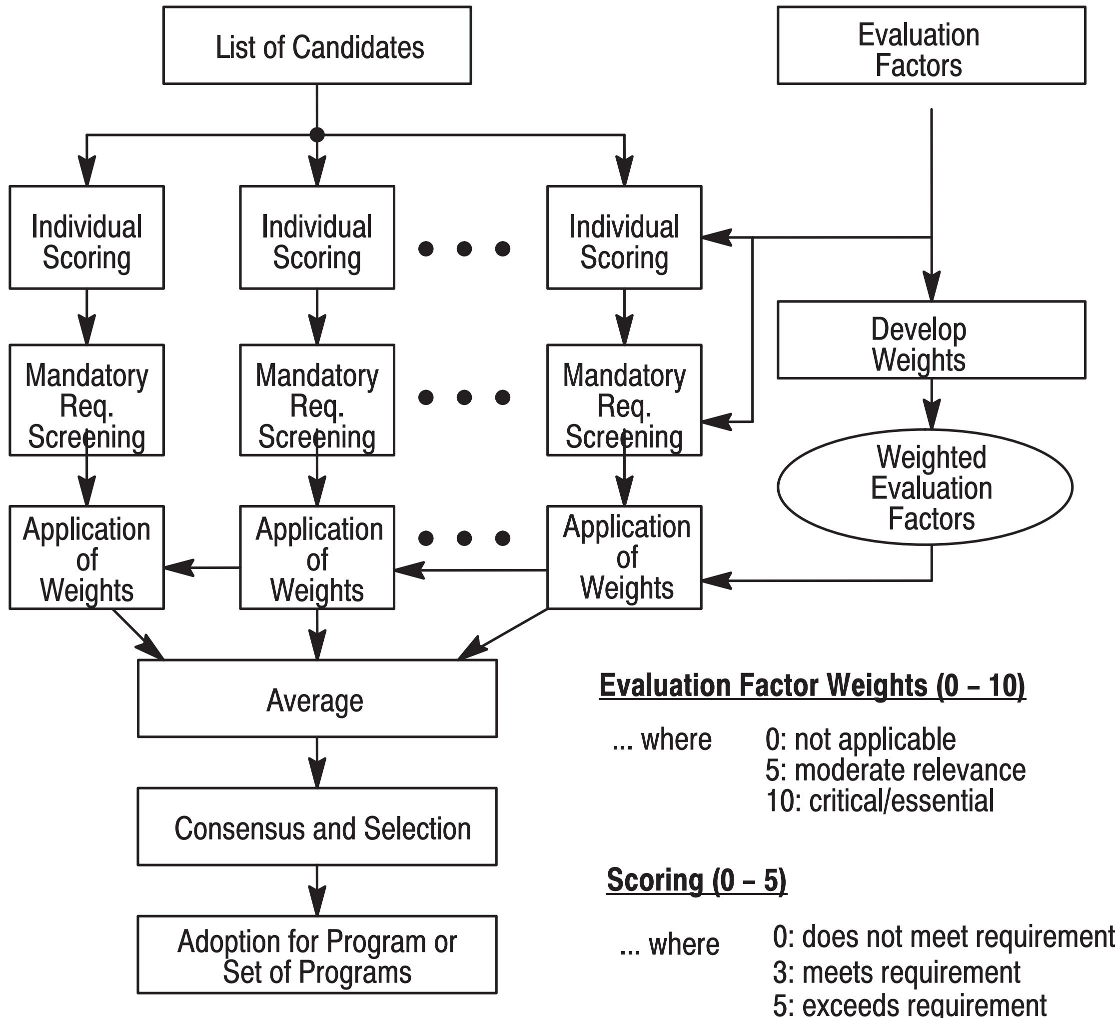


Figure 6 RASSP Interface Standard Selection Process

TECHNICAL	Internal Module	Subsys-tem	Sensor I/O	Inter Chassis	Multi-drop Bus	Test/ Maint Bus
First Tier Criteria						
Bandwidth	X	X	X	X	X	
Physical Constraints	X	X	X	X	X	X
RASSP Compatibility	X	X	X	X	X	X
Real Time	X	X	X	X	X	
Scalability	X			X		X
Second Tier Criteria						
Asynchronous Clocking	X		X	X		
Coherent Mem. Support	X			X	X	
Concurrent Path Scalability	X		X	X		
Data Security	X	X		X	X	
Environmental	X	X	X	X	X	X
Error Correction	X	X	X	X	X	X
Error Detection	X	X	X	X	X	X
Fault tolerance	X	X		X	X	X
Flow Control	X	X	X	X		X
Interconnect Distance	X	X	X	X		
Latency	X	X	X	X	X	
Memory Address Space	X			X		
Message Passing	X	X		X		
Message Size	X	X		X		
Multicast	X			X		
Node Addresses	X			X		X
Reconfigurability	X			X		X
Routing	X		X	X		
Shared Memory	X	X		X	X	
Software Impact	X	X	X	X	X	X
Time Synchronization	X	X		X		
Topology Support	X			X		
Testability Support	X	X	X	X	X	X

Table 1 RASSP Interface Standard Technical Evaluation Factors

7. SUMMARY

This document has presented the specification for RASSP hardware architecture elements. The definition of the Model Year Architecture is based on requirements for signal processor design that have evolved from experience in signal processor system design, as well as the additional requirements imposed by the RASSP philosophy to address a 4X reduction in development cycle and life–cycle costs, technology insertion, upgradability, and extensibility. The characteristics of the Model Year Architecture are that it: 1) must be modular, scalable, and open (non–proprietary), 2) should incorporate non–proprietary standard interfaces wherever possible, 3) should leverage commercial technology whenever possible, 4) must provide a way to incorporate custom components when necessary, 5) must facilitate reusability of components, and 6) must provide support for low–cost hardware and software upgrades for continuous product improvement.

The structure of the reuse library components at the architectural level contains the essence of what makes a resulting architecture a Model Year architecture. Using Object Oriented Design techniques to define the various Functional Architecture constructs provides a convenient and natural approach for developing upgradable architectures as well as designing and maintaining the reuse library elements themselves. Library element object classes include both hardware and software portions which are co–dependent, each of which contributes to the object's total behavior. The hardware and software portions are encapsulated to hide implementation details from the user, limit propagation of changes resulting from upgrades, and to ensure interoperability. The encapsulation implements a standard functional interface that is accessible to the user: a hardware portion called a Standard Virtual Interface, and a software portion that is a standard Application Programming Interface (API). By limiting access to this interface, upgradability and technology insertion is greatly enhanced. In addition, the formal but judicious use of layering within both the hardware and software architecture plays an additional role in defining library element object classes and where various interfaces are most appropriate.

The use of open interface standards will further ensure interoperability between and reuse of various signal processor elements, and will help ensure multiple sources and reasonable costs for compatible interface components, eliminating the dependence on sole–source proprietary components.