RASSP VHDL Performance Modeling Interoperability Guideline

RASSP Contract Number: **DAAL01-93-C-3380**

Honeywell Technology Center

Version: 2.0
February 9, 1996

Table of Contents

1. Introduction

RASSP (Rapid-Prototyping of Application Specific Signal Processors) is a major ARPA/Tri-Service initiative to reinvent the embedded digital signal processor development process. The goal is a four-fold reduction in both time and cost from concept to fielded prototype on both new designs and redesigns.

A primary stage in the RASSP design methodology is detailed system performance modeling. A performance model expressed in VHDL serves as a virtual prototype, aids the identification of both bottlenecks and overdesigns, and supports performance validation. It also enables trade-off studies, analysis, and documentation of design decisions. To facilitate wide spread usage and interoperability of VHDL performance models, a standard convention is required. This document describes the requirements for achieving VHDL performance model interoperability within the RASSP design environment.

Simulation is a comprehensive, cost-effective approach to evaluate the performance of highly complex new designs. VHDL provides a flexible, portable, and fully expressive platform for such performance simulations. Since the flexibility of VHDL can result in incompatible approaches, an interoperability guide is required to achieve interoperability between VHDL performance models developed by different organizations. This document is that guide. A brief background and overview of performance modeling is given before the details of interoperability are discussed. This document uses the Honeywell VHDL Performance Model Library (PML) as a basis for discussion. This document was done under the auspices of the Lockheed Martin Advanced Technology Laboratories (ATL) RASSP program.

1.1 Document Outline

Changes from Version 1.0:

- Version 1.5, in addition to updating the token description, restructured the document. The appendices were removed and captured elsewhere in the PML documentation. The software interface section was removed from the interoperability section, since the software interface is unique to the Honeywell implementation. However, ongoing work within the Lockheed Martin ATL team is focussing on software interface issues and will result in future versions of this document.
- Version 1.6 was virtually identical to version 1.5. Version 1.6 added Figure 4 and had all distribution restrictions removed.
- Changes from 1.6 to 2.0 include further explanation regarding the degrees of interoperability, information about the newly formed VHDL Interoperability Working Group, more detail in the functional memory section, a revised implementation plan, and addition of Internet resources.

The remainder of this document consists of the following sections:

- **"Interoperability Discussion" on page 3:** discusses general interoperability issues, performance modeling definitions, and design process descriptions.
- **"Token Description" on page 12**: describes the signal structure for the interoperable performance model token.
- **"Functional Memory" on page 25**: details a technique for communication of functional information within the bounds of a performance model.
- **"Implementation Plan" on page 31**: contains the plan for future VHDL performance model interoperability guidelines.

- **“Internet Resources” on page 34:** contains the locations of the various documents and code packages described in this interoperability guideline.
- **“References” on page 36:** contains various references and associated reading

This document is also available on-line at: <URL:http://rassp.scra.org/public/tb/honeywell/HONEYWELL-DOCS.html>.

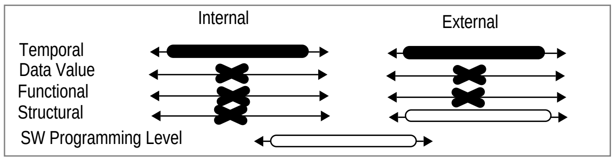
There is also an associated VHDL source code package to go along with this document. The URL for this package is contained in “Internet Resources” on page 34.

2. Interoperability Discussion

2.1 Definition of Terms

The various names associated with modelling abstractions are frequently instance- and application-specific, which can lead to confusion. RASSP is no different than other large distributed projects in this respect. A RASSP taxonomy working group [1] has been formed to address this within the RASSP community. This document, and other associated PML efforts, began development before the working group commenced. This section illuminates some key definitions with respect to hybrid modeling. For a exhaustive model taxonomy, the reader is referred to [1]. This taxonomy is also available at http://rassp.scra.org/public/atl/taxonomy.html. The following definitions are based the RASSP taxonomy work.

- Performance is a collection of the measures of quality of a design relating to the timeliness of the system in reacting to stimuli. Measures associated with performance include response time, throughput, and utilization. A performance model may be written at any level of abstraction. A highly abstract performance model might only resolve the time a multiprocessor cluster requires to perform major system functions, or it can be a less abstract model describing the time required to perform tasks such as memory access of a single CPU. In the context of RASSP, however, the typical abstraction level of a performance model is often at the multiprocessor network level, also called a network architecture performance model. Internal and external data values are not modeled, except for control information. Figure 1 shows the description of performance models using the taxonomy defined by the RASSP working group.



Symbol Key:

●	Model resolves information at specific level relative to Table 1.
▬	Model resolves information at any of the levels spanned, case dependent.
▭	Model optionally resolves information at levels spanned.
▨	Model resolves partial information at levels spanned, such as control but not data values or functionality.
✖	Model does not contain information on attribute.

FIGURE 1. The RASSP taxonomy for performance models

- An uninterpreted model does not model actual data values or data-related functionality either internally or externally. Only control information and control functionality are modeled. These models allow the definition of information flow without interpretation of function. Within the context of the RASSP, using the RASSP taxonomy, leaf cells could be considered performance models while a structure of leaf cells could be considered a uninterpreted model. In some sense an uninterpreted model is something of a superset of performance models. However the distinction between performance and uninterpreted models is too fine to make for purposes of this interoperability document. For purposes of this document, performance and uninterpreted are used interchangeably. Figure 2 shows the RASSP taxonomy for an uninterpreted model.

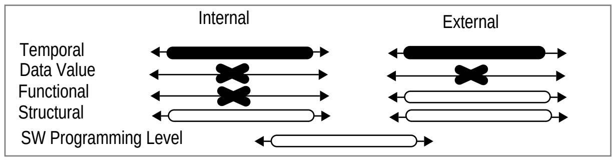


FIGURE 2. The RASSP taxonomy for an uninterpreted model.

- An interpreted model models actual data values and data-related functionality and control-related functionality both internally and externally. Interpreted models can be considered a superset term of functional and behavioral models. For purposes of this document, functional and interpreted are used interchangeably. Figure 3 shows the RASSP taxonomy for an interpreted model.

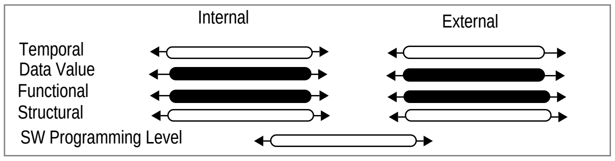


FIGURE 3. The RASSP taxonomy for an interpreted model.

- A mixed paradigm model is composed of both uninterpreted and interpreted models. This type of model is also referred to as a hybrid model.

For background information on VHDL performance modeling, the reader is referred to Aylor, et al. [2] and Kumar [3]. These are excellent references on the basics of performance and hybrid modeling with VHDL. Additional background material may also be found in [5][6][7][8][9].

2.2 Rationale and Methodology Overview

The VHDL performance modeling methodology is targeted towards high level description, specification and performance analysis of computing systems. Figure 4 illustrates the Lockheed Martin ATL RASSP

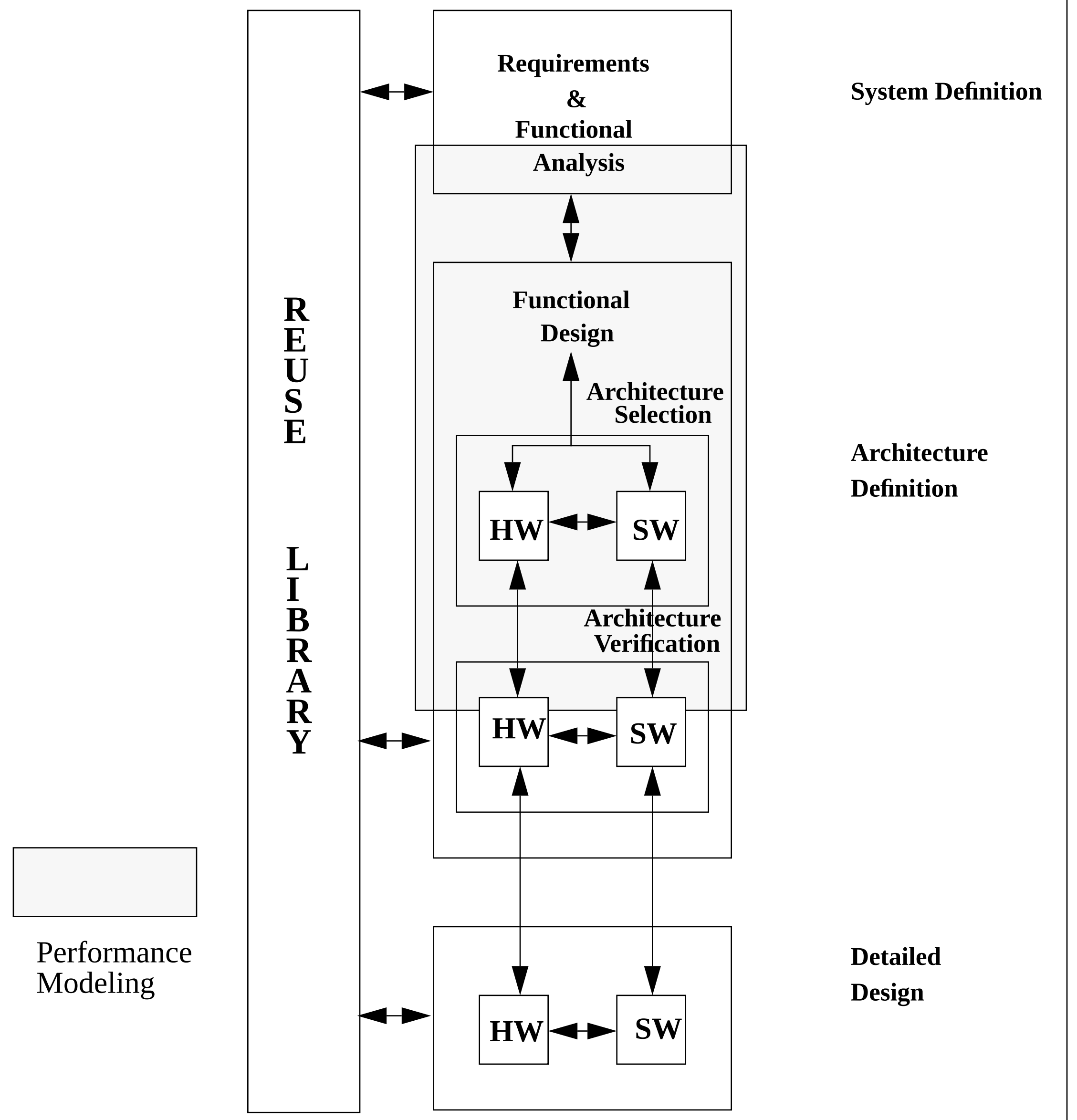


FIGURE 4. The RASSP design process consists of system definition, architecture definition, and detailed design. The shaded area shows where performance modeling may be used in this design process.

design process [4] and where performance modeling fits within the process.

Performance modeling tools and techniques themselves are not targeted towards any particular application. The level at which is appropriate to apply these tools is at defined in Figure 4. The primary use of performance models is airing architecture selection and verification. This includes the actual device or entity under study such as a signal processor, and its environment, such as sensors and actuators. In the case of an electronics system, an architectural level description would include information about both the hardware and software.

Performance modeling provides another tool to the system designer, but is not intended to be stand-alone nor discarded at the end of the system development stage. Performance modeling can aid evaluation of design alternatives, capture design decisions and assumptions, examine system behavior at boundary conditions, and help determine bottlenecks and overdesign. The system designer can also utilize performance modeling for examining system sizing, topology, partitioning and capability issues. An important benefit of performance modeling is that it provides early interaction of system, hardware, and software designers.

Performance modeling and analysis has been done for years. In the mid-70s, N.2 and other languages were developed for performance analysis. There are currently several commercially available software tools for system performance analysis. These include SES Workbench, Bones from the Alta Group of Cadence, Network 2.5, and several others. All these tools provide their own language, capture, and output analysis capabilities. Some also will output VHDL or Verilog. However VHDL provides an excellent platform for performance simulations. The following are some of the reasons for using VHDL for performance modeling:

- VHDL is a standard language and is vendor independent. This is primarily an argument against tool specific formats. Other standard languages that may be used for performance modeling include C and Verilog. However the scope of the performance models presented in this document could not be done within the confines of the Verilog language [10].
- A VHDL performance model provides tight coupling to lower levels of design. Hybrid modeling provides the capability to mix multiple levels of abstraction, which can provide useful, detailed information for part of the design [8]. This is perhaps the most compelling argument for using VHDL for performance modeling.
- VHDL models are easily transported, an important feature for multi-company or division projects, and for projects that span long time periods.
- VHDL is a expressive language with full hierarchy and configurations which allow development and application of highly configurable and flexible models. Consistency and completeness checks are automatic.
- A VHDL model also provides useful documentation of the design and increases the likely of model reuse due to the standard and long term, wide-spread support of VHDL.

VHDL for performance modeling makes sense for designs which will have custom hardware built. It should be kept in mind that for certain applications, such as traditional network communications analysis, some of the above arguments may not be as pressing. In those cases, a commercial network communications modeling tool would be a better solution.

This VHDL performance environment allows the systems architect to capture the system under study in a consistent, verifiable form. The VHDL simulation produces metrics which can be used by any commercial analysis package, spreadsheet, or other appropriate format to aid the design decision process. The results can be directly compared with the system specification to verify that the architecture meets the performance requirements. Once the architecture is verified (the latency, utilization, and throughput meet requirements, the system is self consistent, and size, weight, and power limits are met), the system is ready to proceed to detailed design. The performance model can also produce the architecture's characterization for its use in a higher level model, where this design would just be another building block.

2.3 VHDL Interoperability Working Group (VIWG)

The RASSP program has chartered a VHDL Interoperability Working Group (VIWG). The following discussion is taken from the charter statement of this group, dated January 16, 1996. The reader is encouraged to check the RASSP E&F Web page (http://rassp.scra.org/) for the latest update to the interoperability and taxonomy working groups.

2.3.1 Background

A major contribution to the RASSP 4x improvement goal will be achieved through the re-use of high-level design blocks. These design blocks will come from many sources, so the models must be interoperable for the RASSP 4x goal to be affordably achieved.

A major obstacle to the efficient and effective re-use of VHDL descriptions is the lack of guidelines for the interoperability of higher level VHDL models. To promote the development of interoperable VHDL models, guidelines must be assembled to ensure that only compatible modeling constructs are used for each of the model types needed in the RASSP design process.

2.3.2 Objectives

The goal of the VHDL Interoperability Working Group (VIWG) intends to determine what new guidelines for VHDL model interoperability are necessary for the RASSP design process and to develop those guidelines.The VIWG will also identify existing industry efforts, and promulgate and contribute to those efforts as appropriate for RASSP.

2.3.3 Process

The process for establishing interoperability guidelines can be decomposed into several stages.

1. First we must agree on what we mean by interoperability and what type(s) of interoperability is(are) required in RASSP.
2. Then we must select the distinct model types for which interoperability is required within RASSP and focus on those for which guidelines do not exist.
3. Next, we must establish the necessary model aspects to be covered/specified in the guidelines to ensure interoperability.
4. Finally, we must assemble the guidelines themselves by unifying and selecting the best methods currently in use.

2.3.3.1 Interoperability

Two candidate definitions are posed here. The basic distinction is the effort required to get two or more VHDL models to "interoperate". A related question would be if the interoperability effort should include, or be limited to, things such as type conversions at model interfaces.

(a) Two VHDL models are interoperable if they can be interconnected and used together in the same simulation without modification. [loosely based on IEEE Standard Dictionary, p. 676].

(b) Model Interoperability designates the degree to which one model may be connected to other models, and have them function properly, with a modicum of effort.

Interoperability may be considered between models of the same type or abstraction level, as well as between models of differing types or abstractions. RASSP interoperability efforts must address both of these types of interoperability.

These definitions of interoperability are changing as the working group evolves. These are proposed definitions and the final result will probably be a combination of the proposals.

2.3.3.2 Interoperability Required by RASSP

Since the RASSP design methodology is a top-down approach, and the leverage of reuse is greater the higher the design reused, RASSP should be more concerned about the interoperability of higher-level VHDL models. Also, since industry is already working on many of the issues related to lower-level design reuse, we should monitor/contribute to those efforts, as appropriate, without duplicating them.

Interoperability between certain critical RASSP model combinations should be considered. Such combinations as Token-Based Performance to Virtual Prototype, or Abstract Behavioral to Detailed Behavioral are likely candidates. However, mixed-paradigm interoperability should be considered only after good definitions for homogeneous model interoperability are established. This is because the joint-interoperation (through translator/wrapper generation) of models becomes much easier when a solid and open definition exists for the information content of two different model types,.

2.3.3.3 Model Aspects to be Covered by Guidelines

What is required to make two models interoperable? Those are the aspects that need to be covered by guidelines. Ideally, designers could select VHDL models from a library and focus on interface and behavior (or data and transformations), in a similar manner to pulling functions from a programming library or parts from their bins. This assumes that the designer is making a sensible selection and use of VHDL models.

Model interoperability requires agreement in the following aspects:

1. interface structure
2. interface data format
3. interface timing
4. interface protocol
5. the information content/semantics of exchanged signals

As a next step, we need to define the information content of the respective model types. Preferably this would be done through some sort of concise information model of both the model's internal information content as well as the information contained on the model's interface.

2.3.3.4 Assemble Guidelines

Further pursuit of this requires completion of the above.

2.4 Performance Modeling Interoperability

This document is meant to be a consensus opinion on VHDL performance modeling interoperability. Because Honeywell has a role on the Lockheed Martin ATL team to develop and support VHDL performance models, this document is initially heavily weighted towards that role. Hopefully as this technology becomes more widespread throughout the RASSP community, the interoperability guideline will become more generic. Honeywell is actively interested in any and all feedback.

2.4.1 Same Abstraction Level

Using the VIWG descriptions of interoperability, the bulk of this document is focused towards interoperability of models at the same level of abstraction, specifically at performance models. Figure 5 shows interoperability between the same abstraction level., or horizontal interoperability.

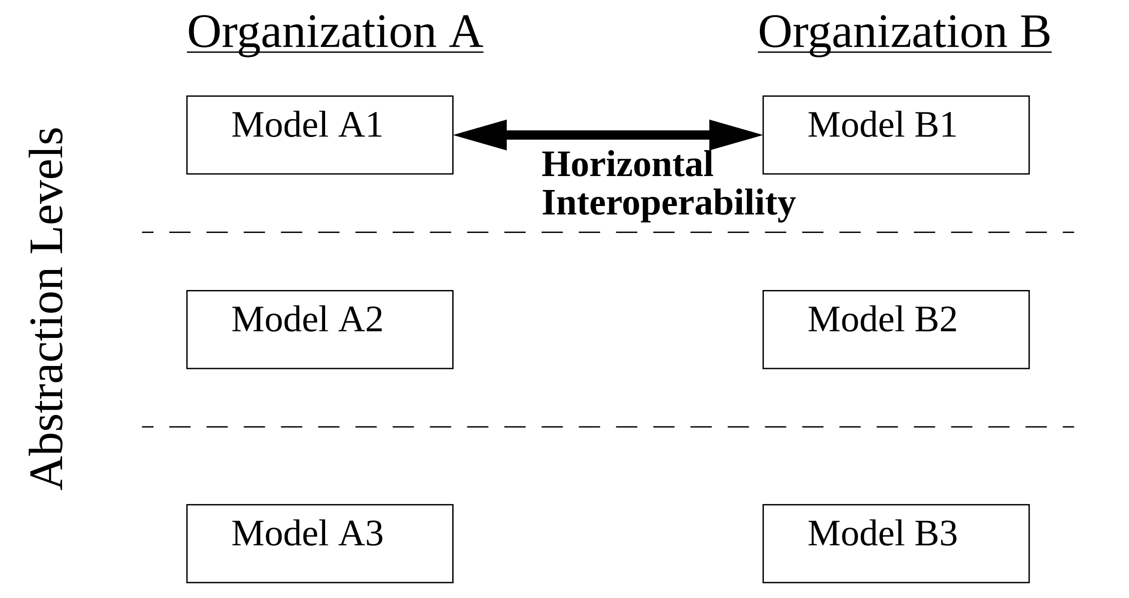


FIGURE 5. Horizontal interoperability is interoperability between models of the same abstraction level from different organizations .

In this case, that interoperability is achieved using a common signal description: tokens. This is illustrated in Figure 6, which shows performance models from the Honeywell library and the UVa ADEPT library working together. This interface example is described in [13].

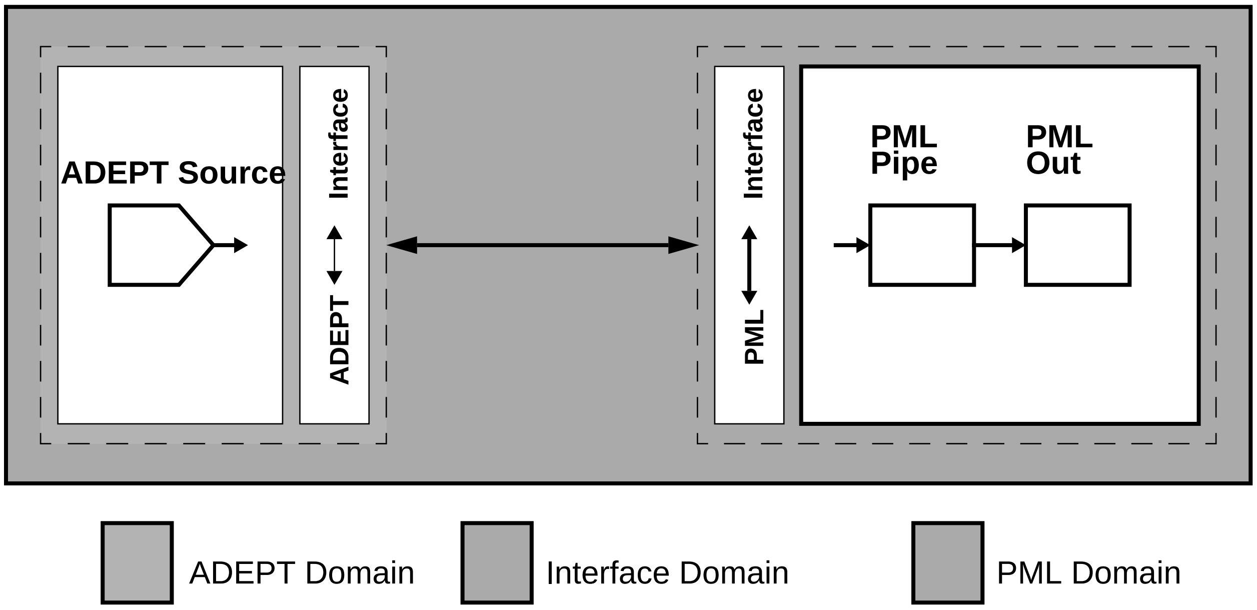


FIGURE 6. Token-based model interoperability between a UVa ADEPT model and a Honeywell PML model. This is an example of horizontal interoperability.

The interface token, which occupies the interface domain in Figure 6, is defined in “Token Description” on page 12.

2.4.2 Different Abstraction Levels

Interoperability between different abstraction levels can happen with performance models. Figure 7 illustrates vertical interoperability, or interoperability between different abstraction levels. Performance mod-

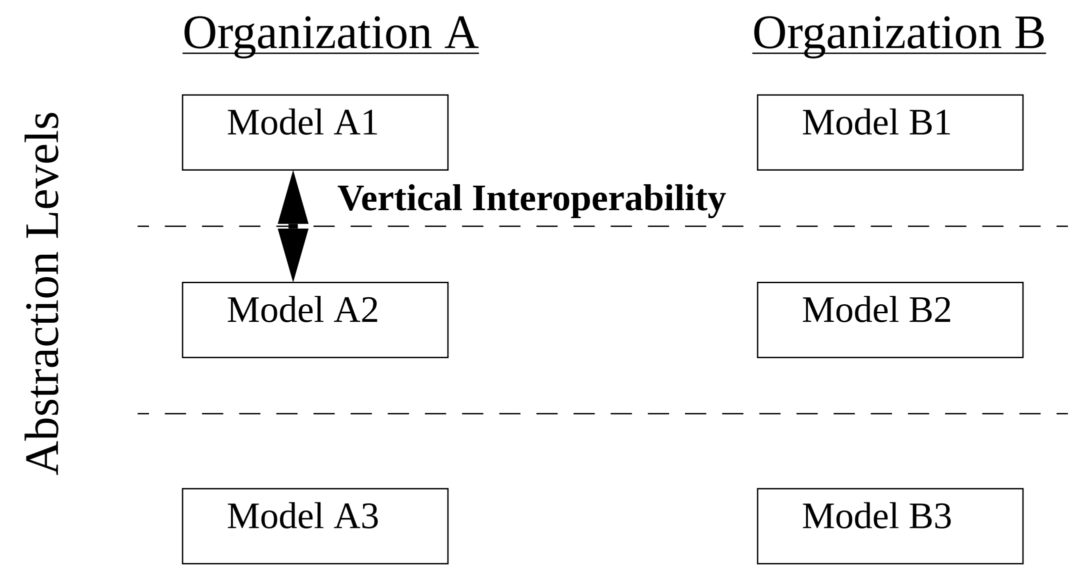


FIGURE 7. Vertical interoperability is interoperability between models of different abstraction levels.

els cover such a broad territory, both within the design process and in terms of representing both hardware and software abstractions, that mixing abstraction levels occurs frequently. Since this obviously makes interoperability much more difficult, this document focuses on token interoperability, as that is the minimum requirement, and is easier to discuss among different organizations.

The section of this manual dealing with interoperability between abstraction layers is the “functional memory.” The functional memory allows a performance model to carry along interpreted data. That interface is well defined and heavily used within the PML although no other VHDL performance modeling approaches currently use it.

Perhaps the most important next abstraction level for performance model interoperability is with software architecture representations. The PML does provide capabilities for modeling software architectures. However there are many approaches, even within the RASSP community, for performance and algorithm analysis of software architectures. Honeywell, MCCI, and Omniview are working together within the Lockheed Martin ATL team to identify a common interface for software representation. That work is ongoing and will be described in future releases of this document.

The reader is also referred to Honeywell Hybrid Modeling Library (HML) documentation [14][15] for further descriptions of mixed level models.

3. Token Description

The token is the basic unit of communication in performance modeling. The token represents uninterpreted data, i.e., the precise value of the data is neither known nor required for the exercising of the model and accumulation of performance statistics. This section defines rules for communication between entities in a system performance model via these tokens.

3.1 Introduction

Signals represent the interconnection of the various elements of the system. The signals within the performance model represent units of information passed from one system element to another. The intent of these signals is to propagate control flow, data flow, and performance information across the model. The paradigm chosen for the current performance modeling environment is based on moving quanta of data, called tokens. While this is different from the typical VHDL paradigm of level sensitive signals, it is a natural technique for modeling abstract behavior with a minimum of effort.

A major goal of the Honeywell performance modeling effort, upon which this guideline is based, was to eliminate some of the major limitations seen in other high level modeling environments. One of those limitations was the restriction of having dedicated point-to-point interconnections; if two components needed to communicate, they needed a dedicated connection between them. This requirement made it difficult to build readily scalable models, or models with flexible topologies. The models developed here allow multiple connection points without requiring the user to expend excessive effort editing of the involved components.

Probably the most significant drawback to modeling of this nature in VHDL is VHDL’s lack of support for dynamic or variant types on signals. As a result, the user has 3 basic options for supporting arbitrarily complex communications:

- Abstract the complex communication to a higher level, more appropriate for the performance model.
- Have one type, but make it a record containing everything one might desire.
- Have specific types for each different communication requirement.

Each of these approaches has its limitations. The first can force “artifacts” into the model which can confuse or cover the model intent. The second will negatively impact simulation performance. The third is perhaps the most costly, as it significantly increases model build time and library complexity, and it limits the ability to easily reconfigure the model to perform trade-off studies.

A token implementation was chosen that combines all of these techniques. Overall, the emphasis is to avoid having separate types so that any component may be connected to any other.

The following section describes the VHDL for the token, and its supporting VHDL definitions. The subsequent section will describe the ways in which this token is used throughout the performance environment. The last section contains the relevant VHDL code.

3.2 Token Description

Section 3.4 on page 18 contains the portions of the VHDL package, token_std-p(b).vhdl, relevant to the token description. This code will be explained bottom up, i.e., each of the types comprising the token will be explained, followed by the description of the entire token.

3.2.1 Initial Constants

First some basic initialization constants are defined. These constants attempt to avoid code obfuscation by allowing the designer to very obviously and cleanly specify that a value is to be set unequivocally to zero.

3.2.2 Supporting Types

The first declared types are the standard integer and real vectors which are so commonly used that further explanation should be unnecessary.

The next type declared is a physical type called `data_size`. This is used to communicate throughout the model the quantity of information involved in a particular transaction. The intent behind this type definition is to allow the various systems designers to work in natural units. For instance, a computer scientist can be entirely willing to describe dataflow in terms of kilobytes of information. On the other hand, an image researcher is possibly more inclined to speak in terms of frames or images. This type allows both representations to coexist, since everything relates at the base level to bits of information. We recommend that some initial size designators are agreed upon, and provide the user with appropriate capability to expand the definition to suit application specific demands.

The next type, `protocol_type`, is an enumerated type containing values for various common and application specific communication protocols. It might be more appropriate to include only standard protocols in this list, with extra slots for user defined protocols.

The `state_type` type defines the different states that the token achieves during generic bus resolution.

`Token_type` enumerates the possible type of tokens which can flow through the system. Certain components, such as memories, can take advantage of this information.

`uGIDType` defines a global type, a subtype of natural, which allows every token to have a unique identifier.

`MAXINSTLENGTH` is a constant that controls the length of instance names in the model, in the `name_type` type. An instance name is the concatenated hierarchical name of a specific component. The technique of hierarchically building instance names allow the unique identification of components within the model hierarchy. Early memory limitations forced us to reduce this constraint to a rather small value, but recent advances in VHDL efficiencies should allow us to increase this to a reasonable value. While an unconstrained value in this case would be most optimal, VHDL does not support such things on signals. This value should be as large as possible so the design is not unduly impacted by having compact naming conventions.

VHDL 93 provides some capability for unique component naming. It is not yet clear whether this will be sufficient to replace the instance naming convention. Due to the lack of generally available VHDL 93 implications at this time, these and other potential advantages from VHDL 93 are not part of this interoperability guideline.

3.2.3 Token Fields

This section describes each token field. The fields are presented here in the categories of control flow, performance, reporting, arbitration, and user parameters.

3.2.3.1 Control Flow Fields

These fields are used by the various performance nodes to model control flow in the design. Each node has an input filter to control the passing of the token. If a token is not allowed pass the filter, no effects from that token are seen within the node. If the token does pass, the node will delay for an appropriate amount of time, and produce a new token.

destination: `name_type.` The destination field on the token contains the instance name of the intended recipient of this token. The various pattern matching capabilities within the performance model allow flexibility with the specification of this field. This is discussed in Section 3.3.1 on page 16.

source: `name_type.` This is generally the instance name of the node which originated this token. It is used, for instance by the memories, to determine whence a request came so that it may be appropriately returned.

t_type: `token_type.` This field indicates the token type. Memory components will use this field to determine whether to return a token (as with a read), or simply consume the incoming token and delay, as with a write.

3.2.3.2 Performance Fields

These fields are the primary carriers of the performance effects in the system. They represent the amount of data that is being transferred between nodes. Each node operates on a particular type of data at a given rate. The above control parameters provide the filters that insure the correct quantum of data is operated upon, and these fields show how much.

size: `data_size` Size or number of this token. For instance, if this were a “memory read’’ token, size would be construed to mean the number of words to read from the memory, and thus indicate to the memory the appropriate time to delay to model the access.

value: `INTEGER` Certain operations require two values to be effective. For instance, reading a 1 MB block from a disk drive does not necessarily require 1 MB worth of read request from the source process. Rather, the system might generate the address range which is to be read, and allocate tables and pointers for storing the requested information. This whole request might consume a few tens or hundreds of bytes of network bandwidth. The returned data would then amount to 1MB worth of data, along with whatever other addressing information is necessary. The value field was established to support this sort of transaction. The outgoing read request will have the data_size filled in with the size of the actual read request packet. The value will contain the amount of data which is to be read from the memory. The memory will receive this token, and delay based upon the length of time it takes to handle the requested amount of data. It will then return a token with the value and data_size fields reversed, so the intervening network (if any) bears the effects of transferring the full read amount.

3.2.3.3 Statistics Fields

These fields are primarily used by the nodes for reporting simulation performance statistics. These values can also be used by the bus resolution function if the other fields prove insufficient for arbitration.

id: `uGIDType` The ID contains a numeric identifier for this token. When a token is created, it receives a global ID number. This global ID is unique between deltas. When a token impinges upon a node, and is then passed through, the token generated by that node inherits the ID of the input token. Tracking of such IDs through the system can then be used to establish the path and various delays that resulted from an initial action.

start_time: `TIME` This is the time that the token was originally created. This field, like the ID, is generally just passed on through performance nodes.

3.2.3.4 Communication Protocol Fields

These fields are used by the bus resolution function and the communication interface elements to handle arbitrated communications. Most signals in the performance models can dispense with the complex arbitration. However, when internal communications becomes a bottleneck and a more precise model is necessary, performance models of communication interfaces and media may be inserted. Those components will make use of the same token, and arbitrate using these fields.

priority: `INTEGER` The message might have a priority associated with it in order to resolve contention, both for the complex arbitration using communication interface models, as well as with the standard model for interconnection. For instance, within the standard interconnection, when a memory only has one connection on it, the memory will increase the priority of the incoming token by one when it returns the requested information, thus insuring that the signal takes the value of the output token.

Within an arbitrated communication line, such as a PI-Bus, the priority can take on whatever behavior is necessary to support the protocol. Since the priority is represented by a VHDL integer, there are at least 32 bits available for encoding complex priority schemes.

state: `state_type` This is the current state of the bus during arbitration handled by communication interface models.

protocol: `protocol_type` controls the way in which the signal will be resolved. One of the simple checks that is performed is to make sure that all tokens on a bus have the same protocol marks.

Other approaches that have been suggested are using bus resolution functions (BRFs) to handle arbitration, and also to have different types for each modelled communication medium, instead of saving this information in a token field.

Bus resolution functions are insufficient to solely handle communications in the cases where timing, such as that for initial acquisition, plays a significant role. This is primarily due to the inability of the BRFs to save state and to consume time.

If the typed approach were taken, editing port and signal types would be necessary to build a model, instead of using the one common interconnect mechanism. That would increase the complexity of the library. Even worse, this would drastically limit the usefulness of the performance model, since configurations would no longer be sufficient for building simulations for performance trade-off analysis of alternative architectures.

3.2.3.5 User Communication Tracking and Control Fields

These fields are provided to allow more flexible modeling and tacking of application specific requirements. Some parameter slots are provided so the user may communicate information from one component to another, for more detailed functional modeling. Other slots are provided to enable in depth analysis of communications behavior.

Except for some internal communications between application software hosted on the Honeywell Processor model, these fields are currently not used.

collisions: `INTEGER` Intended to allow user to track number of collisions a token might have encountered.

retries: `INTEGER` Intended to allow user to track number of retries token has encountered since it began transfer.

route: `INTEGER` Intended to store the initial route a token should take to reach its destination.

This route field is currently described as a single integer value. Multi-stage networks can require non-scalar routing structures. For instance, a routing list is a list of switch-port entries, one for each stage of the switch. Perhaps a dynamic or vector field is appropriate here when those lists cannot be easily encoded into a 32-bit integer. This is an example of where the static nature of VHDL inhibits the ability to easily accommodate arbitrary structures. It might make sense to change all these fields to arrays with limits bounded by deferred constants. However, model-to-model communication would then require additional interoperability guidelines.

3.2.3.6 User Fields

parm1_int: `INTEGER` User specifiable parameter.

parm1_real: `REAL` User specifiable parameter.

parm2_int: `INTEGER` User specifiable parameter.

parm2_real: `REAL` User specifiable parameter.

3.3 Token Usage Conventions

This section describes some of the conventions used with the performance models, based on the above token definition.

3.3.1 Addressing

Token addressing is done by setting the source and destination fields on the token. For instance, if we have a model with one component (Node *A*) generating stimulus for two consuming components, we need to connect all three components with one signal (note that various other high level tools will require point-to-point connections, which then require port modifications, etc.). If the two destination nodes are named *B* and *C*, respectively, then the producing node can alternate between generating tokens with destination fields *B* and *C*.

Input ports on components have filters that accept or reject incident tokens based on a pattern match. This pattern match is a limited regular expression. A regular expression package, regex.vhdl, has been developed by Lt. Greg Peterson of Wright Labs. This package, which is a useful subset of the Unix GNU regular expression package, provides a much greater capability than was previously available in the Honeywell library. Additionally, this package will be available via the normal GNU copyleft license. For example, suppose we start with our above model, but double the receivers so we have A generating stimulus for *B0*, *B1*, *C0* and *C1*. Then if *A* produces a token with destination field *B*?, both *B0* and *B1* will receive it. Likewise will a destination of *C*? be accepted by *C0* and *C1*. This is how multicast (or broadcast, with a destination field of “?*”) is supported.

When a node accepts an incoming token, it has the option of just passing the token straight through (after an appropriate delay) without modification. It can also modify the destination field. It is usually straightforward to piece together a small number of nodes to satisfy quite complex switching requirements, such as crossbars, using this concept of input filters.

3.3.2 Performance

Once a token has been accepted by a node, the node performs some activity based on that input. The generic nodes all model performance based on the incoming token size. Each node specifies a basic unit of work, corresponding to the data size, and a rate at which that work takes place. In this way, each incoming token causes the node to be busy for a time equal to the data size divided by the product of the rate and unit of work. This makes for extremely flexible representation and natural units for each of the different system components.

3.3.3 Arbitration

These tokens and the associated bus resolution functions are designed to minimize the impact to the designer/modeler. In the cases where bus arbitration does not play a significant role in the overall system performance, the various system components may be directly interconnected. If an average value for arbitration overhead is appropriate, delay elements can be inserted. Finally, if arbitration is an issue, then actual communication interface models should be used. However, the same token signal is used throughout to facilitate rapid design entry and reconfiguration.

The model developer also has a choice regarding the desired level of arbitration. The BRF can handle any of following scenarios.

- Lossy communications. Here the highest priority token gains access to the signal. Tokens can be lost, but the intent is to provide good performance. Situations with tuned pipelines, for instance, can use this mode safely.

  State flow: idle = = > busy = = > idle

- Lossless, untimed communication. This mode requires an acknowledge from the receiver. Multiple drivers can be active, but acknowledges will not be lost. Messages may not be broadcast with this configuration, since only single acks can be supported.

  State flow: idle = = > request = = > ack = = > idle

- Lossless, arbitrated, timed communication via communication elements. This is controlled access.

  State flow: idle = = > request = = > ack = = > busy = = > idle

While the primary focus to date on communications network modeling has been packet based and point-to-point communications, modeling a circuit switched network should be quite straightforward to add to the library. This can be done with extending some of the field enumeration types, such as bus_state and token_type, and adding additional capability to the bus resolution function. Additional communication interfaces models would also be developed.

This token passing mechanism is loosely based on the University of Virginia’s (UVa) ADEPT performance modeling scheme [2]. The UVa system uses the third scenario above (lossless, arbitrated, and timed), while this system adds other scenarios.

3.4 Token_std Package

3.4.1 Package Heading

This section has the complete code listing of the Performance Modeling Token_std package, which defines the basic time granularity for the performance model and the token type for communication.

```
-------------------------------------------------------------------------------
--
--  File name  : token_std-p.vhdl
--  Title      : Token Interface Standard Type Definitions
--  Library    : STANDARD
--  Purpose    : Contains basic declarations for performance modeling.
--
-------------------------------------------------------------------------------

PACKAGE Token_Std IS
  --                        CONSTANT DECLARATIONS

  -- Overload ZERO for some predefined types
  CONSTANT ZERO       : INTEGER := 0;
  CONSTANT ZERO_TIME  : TIME    := 0 ns;

  --  Maximum component instance name length
  CONSTANT MAXINSTLENGTH : INTEGER := 81;

  --                           TYPE DECLARATIONS
  -- data_size : a data type representing the size of the token in bytes
  TYPE data_size IS RANGE 0 to INTEGER'high
    UNITS

      -- Basic bit types
      bit_size;
      kbit                  = 1024 bit_size;
      mbit                  = 1024 kbit;

      -- Basic byte types
      byte                  =    8 bit_size;
      kbyte                 = 1024 byte;
      mbyte                 = 1024 kbyte;
      --gbyte                 = 1024 mbyte;--  compiler dependent

      -- Basic machine word lengths
      word_4                =  4 bit_size;
      word_8                =  8 bit_size;
      word_16               = 16 bit_size;
      word_32               = 32 bit_size;
      word_64               = 64 bit_size;

      -- Floating point data types
      fp_4                  = 32 bit_size;
      fp_8                  = 64 bit_size;
      fp_12                 = 96 bit_size;

      -- pixel types
      pixel_6               =  6 bit_size;
      pixel_8               =  8 bit_size;
      pixel_12              = 12 bit_size;
      pixel_16              = 16 bit_size;
      pixel_24              = 24 bit_size;

      -- miscellaneous types
      triangle              = 9 fp_4;
      -- Image types.
      image_512x512x6       = 262144 pixel_6;
      image_512x512x24      = 262144 pixel_24;
      image_1024x1024x6     = 1048576 pixel_6;

    END UNITS;

  -- communication types : A collection of enumerated types to define different
  --                         communication attributes
  TYPE protocol_type IS (none, handshake, pt_to_pt,
                         tmbus,
                         PiBus16, PiBus32);

  TYPE state_type IS (idle, request, ack, busy);

  --  Token Types.  These are the different types of tokens.  This value will
  --  affect the behavior of certain classes of components, such as processors
  --  and memories.
  TYPE token_type IS (DATATOKEN, READTOKEN, WRITETOKEN, CONTROLTOKEN);

  --  String subtype for reading/parsing data from files and generics
  SUBTYPE name_type IS STRING (1 TO MAXINSTLENGTH);

  --  Global Token ID type.  A token with an ID of "0" (uGIDType'LOW) is
  --  considered inactive.
  SUBTYPE uGIDType IS natural;
  TYPE uGID_vector IS ARRAY (NATURAL RANGE <>) OF uGIDType;

  -----------------------------------------------------------------------------
  --  Package    : token_std-p.vhdl
  --  Title      : Resolve Global Identification Tag
  --  Purpose    : Resolution FUNCTION to be used on all SIGNALs of TYPE
  --               Global ID which allows the value to increment.
  --
  --               Function merely returns the highest value driver
  --  Return     : uGIDType;
  --  IN         :
  --     PARAM s uGID_vector
  --               set of driving values
  --  REFERENCES :
  -----------------------------------------------------------------------------
  FUNCTION f_resolve_GID( s : uGID_vector ) RETURN uGIDType;

  --  GIDType:  standard GID TYPE declaration to be used with all Global ID
  --            declarations.
  SUBTYPE GIDType IS f_resolve_GID uGIDType;

  --  Global ID Signal, initialize to active (non zero) value
  SIGNAL GID : GIDType := GIDType'LOW + 1;

  -- uinterface_token : the basic unresolved token type.
  TYPE uinterface_token IS
    RECORD
      --  user fields
      parm1_real     : REAL;            --  these are placed first to avoid
      parm2_real     : REAL;            --  some oddities on Sparcs (ACK!)
      parm1_int      : INTEGER;
      parm2_int      : INTEGER;

      --  control flow
      destination    : name_type;
      source         : name_type;
      t_type         : token_type;

      --  performance fields
      size           : data_size;
```

```
      value          : INTEGER;

      --  token tracking or statistics fields
      id             : uGIDType;
      start_time     : TIME;

      --  communication fields
      priority       : INTEGER;
      state          : State_Type;
      protocol       : Protocol_Type;

      --  user communication tracking and control fields
      collisions     : INTEGER;
      retries        : INTEGER;
      route          : INTEGER;

    END RECORD;

  ----------------------------------------------------------------------------
  --  Package    : token_std-p.vhdl
  --  Title      : Token Less Than
  --  Purpose    : token-less-than for arbitration resolution
  --  Return     : BOOLEAN;
  --  IN         :
  --     PARAM l,r utoken
  --               pair of tokens to compare
  --  REFERENCES :
  ----------------------------------------------------------------------------
  FUNCTION "<" ( l, r : uinterface_token ) RETURN BOOLEAN;

  -- utoken_vector : an array of utoken types. To be used in declaring
  --                    signal arrays of utoken types
  TYPE uinterface_token_vector IS ARRAY (NATURAL RANGE <>) OF uinterface_token;

  ----------------------------------------------------------------------------
  --  Package    : token_std-p.vhdl
  --  Title      : Resolve Token
  --  Purpose    : Resolution FUNCTION to be used on all SIGNALs of TYPE
  --               utoken.  Highest priority (based on "<" function) wins
  --               arbitration.
  --  Return     : utoken;
  --  IN         :
  --     PARAM s utoken_vector
  --               set of driving values
  --  REFERENCES :
  ----------------------------------------------------------------------------
  FUNCTION f_arbitrate_bus( s : uinterface_token_vector )
    RETURN uinterface_token;

  -- Token:  standard token TYPE declaration to be used with all signal
  --         declarations.
  SUBTYPE interface_token IS f_arbitrate_bus uinterface_token;

  -- token_vector:  An array of standard token values.  To be used in declaring
  --                SIGNAL arrays of tokens.
  TYPE interface_token_vector IS ARRAY(NATURAL RANGE <>) OF interface_token;

  --  Default token value (cleaner than having the "U" values everywhere)
  CONSTANT init_interface_token : uinterface_token := (
     collisions     => ZERO,
     id             => ZERO,
     parm1_int      => ZERO,
     parm1_real     => 0.0,
     parm2_int      => ZERO,
     parm2_real     => 0.0,
     priority       => ZERO,
     protocol       => none,
     retries        => ZERO,
     route          => ZERO,
     size           => 0 bit_size,
     start_time     => ZERO_TIME,
     state          => idle,
     t_type         => DATATOKEN,
     value          => ZERO,
     source         => (OTHERS => NUL),
     destination    => (OTHERS => NUL)
     );

END Token_Std;
```

3.4.2 Package Body

```
-------------------------------------------------------------------------------
--
--   File name :  token_std-b.vhdl
--   Title     :  Token Interface Standard Type Definitions
--   Module    :  STANDARD
--
-------------------------------------------------------------------------------

PACKAGE BODY Token_Std IS
  -----------------------------------------------------------------------------
  -----------------------------------------------------------------------------
  --                             FUNCTIONS
  -----------------------------------------------------------------------------
  -----------------------------------------------------------------------------
  FUNCTION "<" ( l, r : uinterface_token ) RETURN BOOLEAN IS
  BEGIN

    --  cascaded tests to minimize number of tests to perform.  This is a
    --  pretty extensive list, though not exhaustive.  The tradeoff is of
    --  course performance vs completeness.
    IF (l.priority /= r.priority) THEN
      RETURN (l.priority < r.priority);
    ELSIF (l.id /= r.id) THEN
      RETURN (l.id < r.id);
    ELSIF (l.t_type /= r.t_type) THEN
      RETURN (l.t_type < r.t_type);
    ELSIF (l.size /= r.size) THEN
      RETURN (l.size < r.size);
    ELSIF (l.value /= r.value) THEN
      RETURN (l.value < r.value);
    ELSIF (l.retries /= r.retries) THEN
      RETURN (l.retries < r.retries);
    ELSIF (l.collisions /= r.collisions) THEN
      RETURN (l.collisions < r.collisions);
    ELSIF (l.protocol /= r.protocol) THEN
      RETURN (l.protocol < r.protocol);
    ELSIF (l.destination /= r.destination) THEN
      RETURN (l.destination < r.destination);
    ELSIF (l.source /= r.source) THEN
      RETURN (l.source < r.source);

    ELSE
      RETURN FALSE;
    END IF;

  END "<";
```

```
------------------------------------------------------------------------------
--
--   FUNCTION name: f_arbitrate_bus
--   parameters:    IN         s          -- utoken_vector
--   RETURNs:       utoken   -- resolved value based on driving values.
--   purpose:       Resolution FUNCTION for the token  system.
------------------------------------------------------------------------------

FUNCTION f_arbitrate_bus( s : uinterface_token_vector )
  RETURN uinterface_token IS

  ----------------------------------------------------------------------------
  --  placeholders for drivers that are busy, ack, idle, or request.
  ----------------------------------------------------------------------------
  VARIABLE Busy_token     : uinterface_token;
  VARIABLE Ack_token      : uinterface_token;
  VARIABLE Idle_token     : uinterface_token := init_interface_token;
  VARIABLE Request_token  : uinterface_token;

  ----------------------------------------------------------------------------
  --  Booleans to track the different driver states
  ----------------------------------------------------------------------------
  VARIABLE current_protocol : protocol_type;      --  resolved bus protocol
  VARIABLE current_state    : state_type := idle; --  current state of bus
  VARIABLE current_driver   : INTEGER := s'LOW;   --  current driving signal

BEGIN
  --  If no drivers (?) just drive initial token
  IF (s'LENGTH < 1) THEN
    RETURN init_interface_token;

  -- If one input, default is single driver
  ELSIF (s'LENGTH = 1) THEN
    RETURN s(s'LOW);

  ELSE
    --  This is for ensuring all drivers have the same protocol.
    current_protocol := s(s'LOW).protocol;

    poll_drivers: FOR i IN s'RANGE LOOP
      CASE current_protocol IS

        --------------------------------------------------------------------
        --  Until other methods are provided, they all use the same
        --  resolution scheme.
        --------------------------------------------------------------------
        WHEN OTHERS =>

          ------------------------------------------------------------------
          --  Ensure all drivers have the same protocol. If not, this is a
          --  fatal error.  Perhaps this could be a compile time check?
          ------------------------------------------------------------------
          IF (current_protocol = none) AND (s(i).protocol /= none) THEN
            --  inherit protocol if current is "none"
            current_protocol := s(i).protocol;
          END IF;

          IF (s(i).protocol /= none) THEN
            ASSERT s(i).protocol = current_protocol
              REPORT "BUS PROTOCOLS NOT ALL THE SAME"
                SEVERITY error;
          END IF;

          ------------------------------------------------------------------
          --  Simple priority based arbitration for concurrent requests.
          --  State flow:
          --   idle ==> request ==> ack ==> busy ==> idle  (full handshake)
          --   idle ==> request ==> ack ==> idle           (acknowledge only)
          --   idle ==> busy ==> idle                      (lossy)
          ------------------------------------------------------------------
          --  Note that standards of information hiding, etc, have been
          --  slightly ignored for this function.  Since the brf is
          --  potentially one of the larger drains on simulation
          --  performance, we have chosen to not make extensive use of
          --  function calls.  The cases are also not syntactically
          --  parallel, nor are they necessarily even in the order of the
          --  type declarations.  By design for performance.
          ------------------------------------------------------------------
          --  BRF must handle resolution of the following scenarios:
          --    (1) lossy communications, where the highest priority token
          --        gains access to the signal.  Tokens can be lost, but the
          --        intent is to provide good performance.  Situations with
          --        tuned pipelines, for instance, can use this mode safely.
          --    (2) lossless, untimed communication.  This mode requires an
          --        acknowledge from the receiver.  Multiple drivers can be
          --        active, but acknowledges will not be lost.  Messages
          --        may not be broadcast with this configuration, since
          --        only single acks can be supported
          --    (3) lossless, arbitrated, timed communication via
          --        communication elements.  This is controlled access.
          ------------------------------------------------------------------
          CASE current_state IS
            WHEN idle    =>
              --------------------------------------------------------------
              --  pick first encountered non-idle state
              --------------------------------------------------------------
              IF (s(i).state /= idle) THEN
                current_driver := i;
                current_state  := s(i).state;
              END IF;
            WHEN request =>
              CASE s(i).state IS
                WHEN idle    =>
                WHEN request =>       --  ccncurrent requests can always
                                       --  occur.  The highest priority
                                       --  request wins arbitration
                  IF (s(current_driver) < s(i)) THEN
                    current_driver := i;
                  END IF;
                WHEN ack | busy =>    --  ack and busy always win over
                                       --  request.
                  current_driver := i;
                  current_state  := s(i).state;
              END CASE;               --  s(i).state
            WHEN ack     =>
              CASE s(i).state IS
                WHEN idle    =>
                WHEN request =>
                WHEN ack     =>
                  ----------------------------------------------------------
                  --  Multiple Acks ignored, but perhaps should be warned
                  --  against since that means you're doing a multicast with
                  --  a point-to-point scheme.  Might lose part of a token
                  --  that way,  since there is no mechanism to  handle
                  --  multple acknowledges.
                  --
                  --  Should we rather arbitrate to have deterministic
                  --  results? Likely not worth the effort, since the ack
                  --  can only be from the previous request on the
                  --  line (assuming correct driver behavior).
                  ----------------------------------------------------------
```

```
                    --ASSERT FALSE
                    --  REPORT "Multiple acknowledges on signal"
                    --    SEVERITY WARNING;
                  WHEN busy    =>         --  busy always wins over ack
                    current_driver := i;
                    current_state  := busy;
                END CASE;                 --  s(i).state
              WHEN busy    =>
                CASE s(i).state IS
                  WHEN idle    =>
                  WHEN request =>
                  WHEN ack     =>
                  WHEN busy    =>         --  Multiple busys will occur with
                                          --  lossy communications.  The
                                          --  highest priority driver wins
                    IF (s(current_driver) < s(i)) THEN
                      current_driver := i;
                    END IF;

                END CASE;                 --  s(i).state
            END CASE;                     --  current_state
        END CASE;                         --  current_protocol
      END LOOP;                           --  poll_drivers
      RETURN s(current_driver);
    END IF;                               --  multiple drivers
  END f_arbitrate_bus;

  -------------------------------------------------------------------------------
  --   FUNCTION name: f_resolve_GID
  --   parameters:
  --       IN         s         -- ugid_vector
  --   RETURN:        ugid      -- resolved value based on driving values.
  --   purpose:       Resolution FUNCTION for the Global ID system.
  -------------------------------------------------------------------------------
  FUNCTION f_resolve_GID( s : uGID_vector ) RETURN uGIDType IS
    VARIABLE result         : uGIDType := uGIDType'LOW;
  BEGIN
    -- If no input, initialize to first active (non zero) value
    IF (s'LENGTH < 1) THEN
      RETURN uGIDType'LOW+1;

    --  If it's length one, return that driving value.
    ELSIF (s'LENGTH = 1) THEN
      RETURN s(s'LOW);

    --  Return the highest value
    ELSE
      ResolveGID : FOR i IN s'RANGE LOOP
        IF (result < s(i)) THEN
          result := s(i);
        END IF;
      END LOOP ResolveGID;
      RETURN ( result );
    END IF;
  END f_resolve_GID;

END TOKEN_STD;
```

4. Functional Memory

This section details the capabilities and usage of the functional memory as a mechanism to facilitate functional information flow within a performance model.

4.1 Rationale

A model of a system that focuses purely on the performance aspects of the system without any detailed information regarding expected control is necessarily limited. There can arise situations where specific situations, perhaps boundary conditions, must be explicitly tested. In other cases, stochastic stimulation of the system is insufficient, especially if well-characterized application-dependent behavioral information is unavailable. At such times, the designer will naturally desire the ability to model more detailed system functionality in the model as a mechanism for better determining the performance. In such situations, the functional detail exists in the model for the sole purpose of providing finer grain stimulation capabilities within the model.

Therefore, a mechanism to communicate functional information within a performance model is both appropriate and necessary. Ideally, a mechanism is desired by which system components (both hardware and software) can exchange the desired information *without* significant effort on the part of the model developer. The communication mechanism should also not create significant model artifacts that occlude the original intent and form of the model.

4.2 Requirements

The functional communication mechanism should have the following characteristics:

- **Flexibility:** The mechanism should be sufficiently flexible to permit the interconnection of an arbitrary number of components. It should allow different types of data to be communicated. Arbitrary sizes of communications should be supported.
- **Usability:** The user should not have to build complex interfaces to use this communication mechanism. A straightforward, standardized interface should be available. It should not be necessary for the designer to extensively modify a design or add modeling artifacts to utilize this capability. In particular, addition of entity ports and threaded, hierarchical signals should not be a requirement. However, such an approach should remain possible.
- **Reliability:** The mechanism for data transfer should be deterministic. It should report errors and conflicts.
- **No Resource Utilization:** The functional communication mechanism should not consume resources that would affect performance characteristics. In particular, the transfer of information should not in and of itself consume simulated time.
- **Persistence:** The communication mechanism should be persistent, so that data can be transferred across time, without having to explicitly synchronized the transmitters/receivers.
- **Good Performance:** Model performance should not be significantly adversely affected by the functional communication. In particular, if the mechanism is unused, there should be no noticeable affect to runtime performance. If it is used, performance should be reasonable. Everything being relative, putting bounds on it is difficult at best. We have an objective, though not a requirement, of linear per-

formance degradation with respect to the number of taps on the communication, and the number of types of supported memories.

- **Clean Implementation:** The mechanism should also be wholly within the specified VHDL language, deterministic, and portable to a variety of VHDL environments.

4.3 Language Limitations

Unfortunately, VHDL is not well suited to such communication due to strongly typing and in general does not support dynamic information. In particular, the inability to pass pointers (access types) on signals, and lack of variant records, inhibits the ability to exchange arbitrary information. This means that any general VHDL mechanism will have fundamental limitations.

Global variables, once they are formalized and a real part of the VHDL standard, will accommodate this capability. With care, they can supplant the functional memory approach presented here.

4.4 Application Programmer’s Interface (API)

Given the above requirements and an acknowledgment of VHDL limitations, we now present a solution. This approach is based on a memory component, a complex VHDL signal, bus resolution function, and interface routines accommodating the transfer and exchange of functional information.

4.4.1 Functional Memory

To accommodate the persistence requirement, an actual memory element must be instantiated in the model. This memory, shown in Figure 8, appears as a shared system resource, to which an arbitrary num-

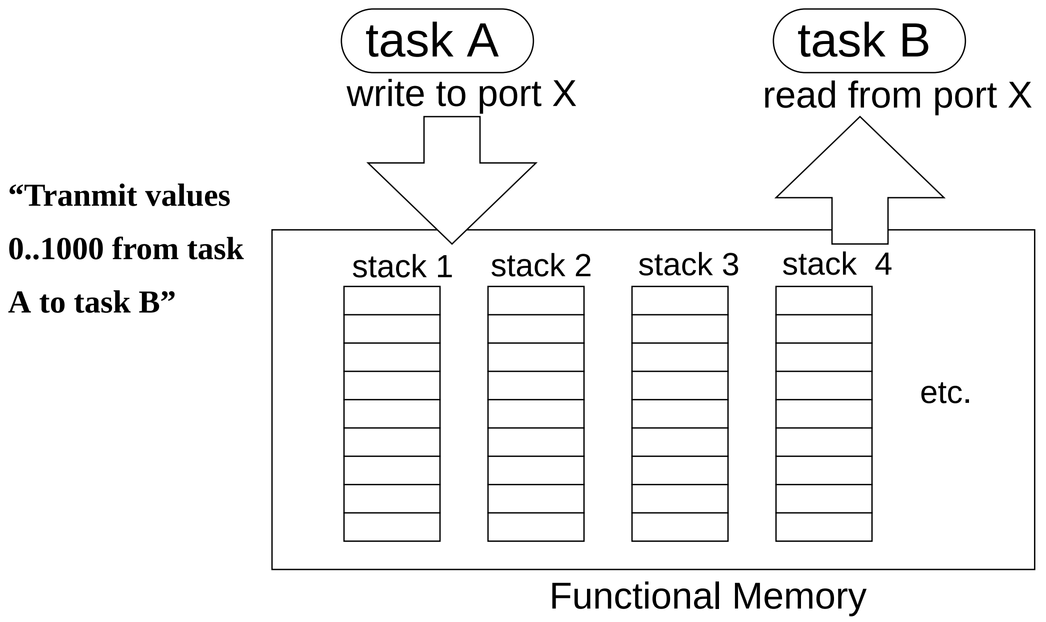
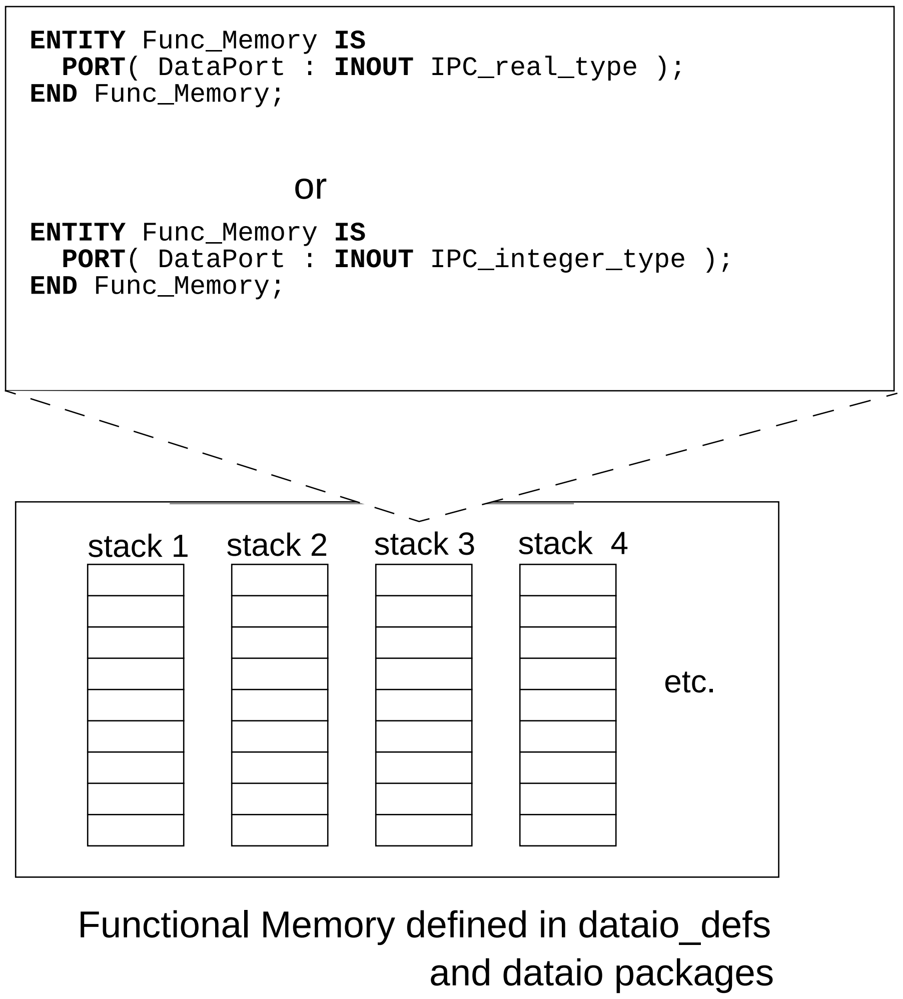


FIGURE 8. Functional Memory component allows software tasks to exchange large amounts of data without impacting simulation time (no resource utilization)

ber of modules can be connected. The memory element has one port, Dataport, as shown in Figure 9. The



```
ENTITY Func_Memory IS
  PORT( DataPort : INOUT IPC_real_type );
END Func_Memory;
```

or

```
ENTITY Func_Memory IS
  PORT( DataPort : INOUT IPC_integer_type );
END Func_Memory;
```

stack 1
stack 2
stack 3
stack 4
etc.
Functional Memory defined in dataio_defs and dataio packages

FIGURE 9. Functional memory entity

memory is designed for simulation performance, and to support a large number of connections. However, the nature of VHDL bus resolution functions can degrade performance under certain situations. For instance, consider the case where 100 components are connected, and 2 of them communicate wildly, while the rest communicate but rarely. In this case, if better simulation performance were necessary, those two which communicate heavily could have their own functional memory instance. The inner workings of the memory are not detailed herein, but are intended to be transparent to the user. However, the connection to that memory via the signal must be explained.

4.4.2 Signal Instantiation

In Figure 10 we see the signal type `IPC_integer_type`. This signal contains fields for arbitration, as well as the fields to accommodate the actual data transfer. The data transfer is actually handled using packed arrays. In essence, the user places the data to be transferred into an array, and the functional memory and associated functions handle moving that data to and from the memory. The length of the array is controllable by the user, but such control is only required for simulation performance optimization, and does not significantly impact modeling. The type of data to transfer can be translated to integer form, transmitted, received, and then translated back to the original form from integers. This translation step is necessary because VHDL is strongly typed. It is a trivial matter to create additional signal types, arbitration functions, and memory elements by simply changing the array type. For instance, much of our current data is of type REAL. Instead of writing translator functions, we just created instances of the appropriate routines so that they handle real arrays instead of integer arrays. Conceivably, record and text arrays could also be developed for higher performance transfers of more complex objects. However, the intent of this package is to allow a few basic transfer types, (integer vector and real vector), and have the user pack and unpack their records into those vectors. This will keep the number of memories and signal types at a minimum, to avoid clutter in the model.

To use the functional memory, one must first make an instance of the memory. Next, the model must have one signal which connects that memory to all the modules which would communicate via this memory. The bus resolution function and various interface routines will handle all the necessary arbitration on this signal. In this way, users need not concern themselves with intricate routing, etc. One must simply add the memory, and hook a signal to it.

Two distinct modeling styles are supported. One can make use of explicit signals and ports, and connect the communicating entities and the functional memory together. The other approach is to use a global signal, so that model artifacts are minimized. Of course, combinations of these are encouraged as the models indicate. We have found the cleanest approach is to declare this one signal in a package, as shown in Figure 10. Then if a module desires to use the memory, one need only include the package in a VHDL use statement. No explicit ports are necessary, so the model can remain “pure” from modeling artifacts.

```
LIBRARY GEN;
USE gen.dataio.IPC_integer_type;

PACKAGE ioport_signal IS
  SIGNAL IOPort: IPC_integer_type;
END ioport_signal;
```

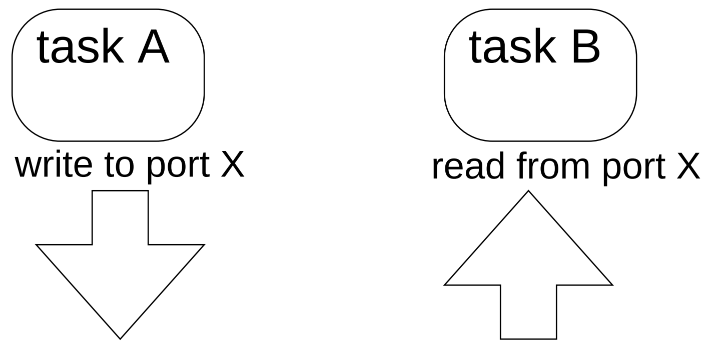
FIGURE 10. Functional Memory Signal Package

The IPC_X_Type Beside understanding the basic type for the data transfer, the remainder of the signa; structure is intended to be transparent to the user.

4.4.3 Read/Write Routines

So far we’ve covered the memory and the signal to communicate with the memory. Next is the way in which the user will use the memory. We decided to provide two simple routines, read and write, through which the memory is accessed. The specification for these functions can be seen in Figure 11. The default values allow trivial usage of the memory without complex user interaction. For instance, the basic command to transfer an integer vector of arbitrary length to the functional memory is:

`write(IOPort, my_array);` -- assumes we use the above defined IOPort signal.



```
PROCEDURE write (SIGNAL ioport: inout IPC_integer_type;
                        item:   in integer_vector;
                        ID:    in memory_identifier_type := 0;
                        offset: in natural := 0;
                        count:  in natural := 0
                        );

PROCEDURE read  (SIGNAL ioport: inout IPC_integer_type;
                        item:   out integer_vector;
                        ID:    in memory_identifier_type := 0;
                        offset: in natural := 0;
                        count:  in natural := 0
                        );
```

FIGURE 11. Functional Memory Access Routines

To read that same information back out of the memory is equivalently trivial for any other element connected to the IOPort signal (explicitly or via the VHDL use statement):

`read(IOPort, my_array);` -- assumes we use the above defined IOPort signal.

The parameters “ioport” and “item” are required parameters. The first is the signal over which the item is to be transferred. In this way, memory of different basic types or multiple memories of the same type but split for performance reasons (as described above) can be supported. The item is an integer vector, and can be of arbitrary length. If the basic length supported by the IPC_type is too small to accommodate the whole vector, the routines are smart enough to establish a link to stream the data to/from the memory without requiring the user to handle each transaction. As was mentioned above, the user might want to modify the size of the internal array to optimize simulation performance. This can be done by changing the value in the dataio_defs package shown in Figure 10, which is intended to be visible to the user.

```
PACKAGE DataIO_defs IS
  CONSTANT IPC_BUFFER_SIZE : INTEGER;--  deferred constant
  SUBTYPE memory_identifer_type IS INTEGER;
END DataIO_defs;

PACKAGE body DataIO_defs IS
  CONSTANT IPC_BUFFER_SIZE : INTEGER :=  4096;
END DataIO_defs;
```

FIGURE 12. Functional Memory Signal Package

The ID field is to differentiate between the various types of memory transfers which can take place. One view could be that the ID describes a mailbox. At each ID then, data is stored, and can be retrieved. The ID is currently an integer. Integers provide the user with 2^{32} different memory types, so this is likely sufficient, unless the user decides to use bitfield identifiers. Another approach is to use an enumerated type. In any case, the basic type for the memory ID is intended to be user accessible. Use it at will.

One limitation with the functional memory is that there may not be concurrent accesses made to identical IDs within the same delta. read-read, read-write, and write-write are NOT in any way supported. Note that such accesses cab be supported at the same simulation time, just not with overlapping deltas. Since each transfer will necessarily consume multiple delta, if the user desires to make use of the same-time multiple access capability, the user must provide some external synchronization mechanism. Note also that concurrent accesses to different IDs is fully supported.

Within each ID in the functional memory, data is stored at particular offsets or addresses. This provides the user with the ability to initially send, for instance, a large frame of data. Subsequent writes might only affect a small portion of the saved data, so the user could send small blocks of data to overwrite only those intended portions. This should increase simulation performance. Note that the granularity of these offsets is limited to that of the `IPC_BUFFER_SIZE` defined above.

Finally, the count field allows the user to transfer portions of the memory. The default is that count equals the length of the array passed into the read/write function. If smaller portions are desired and specified, the transfer will consumer fewer simulation resources.

5. Implementation Plan

Indoctrination of the modeling guidelines is the overall responsibility of the Lockheed Martin ATL RASSP team. This section defines the plan to facilitate the use of the guidelines. Achieving consensus on performance modeling interoperability will be a long term process. This interoperability guideline is just a step in that direction.

The first step is reaching consensus on the definition and requirements for performance modeling. While this step has not yet completely finished, good progress has been made. There is an ongoing activity, coordinated by the Education and Facilitation RASSP contractor, to coordinate the various RASSP modeling efforts and to reach a common taxonomy [1].

In addition to reaching consensus on definition and requirements, individual organizations must take the status necessary to achieve interoperability. These responsibilities include model development, benchmarks/demonstration development, and tool interface modifications. Honeywell has enhanced the VHDL performance model library to support the type of RASSP designs being done by ATL. Honeywell delivered this enhanced library, Version PML_04a, to ATL in January 1996. Part of the development plan also calls for further work on interfaces to the hardware and software architecture tools. Honeywell is relying on standard interfaces as much as possible, however much interoperability work remains in this area.

5.1 Building Interoperable Performance Models

Lockheed Martin ATL used the token described in "Token Description" on page 12 in their virtual prototype for benchmark #1. Results of this effort are described in [12]. Figure 13 shows the architecture of the

Synthetic Aperture Radar, which was the basis for benchmark #1. This shows the level at which performance modeling is useful.

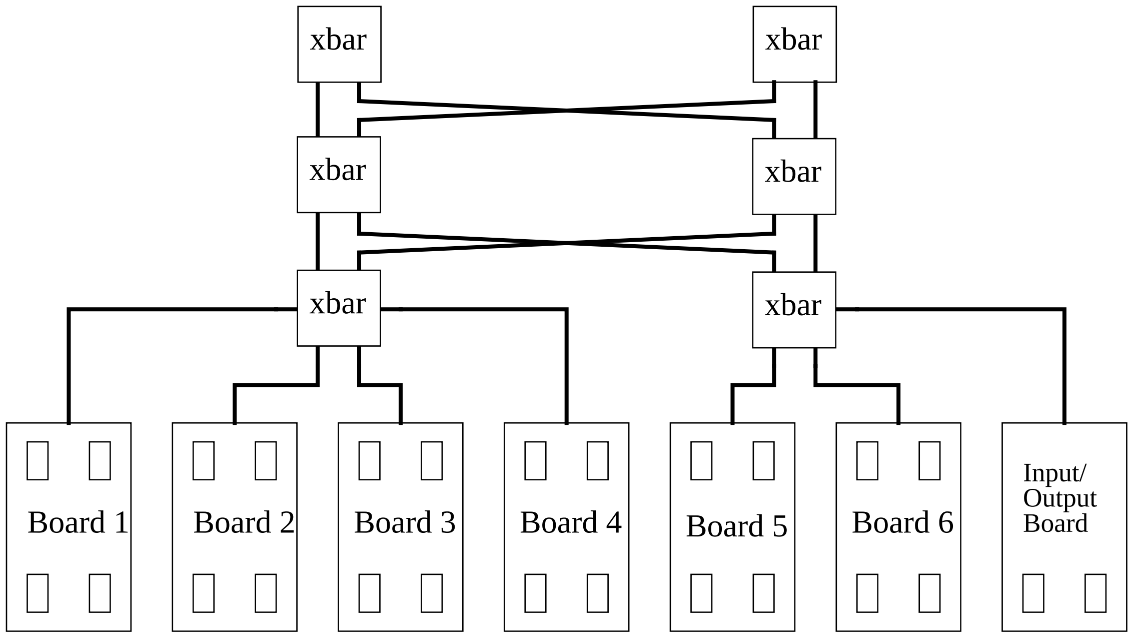


FIGURE 13. The ATL SAR architecture for RASSP Benchmark #1. The performance model of this architecture utilized the interoperability token along with custom performance models.

The University of Virginia has developed an interface between the UVa ADEPT performance models and the Honeywell PML performance models using the interoperability token [13]. UVa has a working model of this interface, which was shown in Figure 6.

The interoperability package will be distributed to all RASSP contractors who need or develop performance models. This package requires no license agreement. The location of this package is described in “Internet Resources” on page 34.

In addition to the interoperability package, several other packages are recommended which should ease the development of interoperable VHDL performance models. The location of all packages described below is defined in “Internet Resources” on page 34.

- The VHDL Strings Package was developed by Lt. Greg Peterson of Wright Labs. This is a string manipulation package written in VHDL. Regular expression routines are also provided.
- The IEEE VHDL Math package. This contains the math routines essential for stochastic analysis.
- Random Number Generator package.
- The GDL (Generic Distribution Language) package. This is a Honeywell-developed package that allows the use of generics for describing complex stochastic distributions. This package has some restrictions on its use but is freely available.

Honeywell periodically delivers a release of the PML to Omniview for wider distribution. The current PML library release number is PML_04a. The Honeywell PML is available to RASSP contractors. Honeywell will deliver the library to our direct customers. Others may obtain the library, under a no-cost license agreement, from Honeywell’s commercialization partner, Omniview. For contact information see “Internet Resources” on page 34.

5.2 Future Releases of the Interoperability Guideline

Future releases of this document will concentrate on two areas: further refinement of the token definition and the definition of a software architecture interface. Current plans call for Version 3.0 of this document to be released in July, 1996. The integration effort between ATL, MCCI, Omniview, and Honeywell will be well underway at that point. A working interface definition should be available by July, 1996. This interface definition will take the form of API service calls to a performance processor model.

Further work by the VHDL Interoperability Working Group will also have an effect on this document. However it is to early to say what that will be. The definition, and extent, of desired RASSP interoperability is expected to be defined by that working group. Those results will likely cause changes to the definitions and formats contained in this document.

Current program plans call for one remaining release of this document, Version 3.0.

6. Internet Resources

This section summarizes where related documents and packages reside on the Internet.

- RASSP E&F home page:

  http://rassp.scra.org/

- Honeywell Technology Center RASSP page:

  http://www.htc.honeywell.com/projects/rassp/

- Lockheed Martin Advanced Technology Laboratories page:

  http://192.35.37.25/rassp_web/

- University of Virginia RASSP page:

  http://csis.ee.virginia.edu/~rassp/

- RASSP VHDL Modeling Terminology and Taxonomy:

  http://rassp.scra.org/public/atl/taxonomy.html

- Interoperability document (this document):

  http://rassp.scra.org/public/tb/honeywell/HONEYWELL-DOCS.html

- Interoperability VHDL package:

  ftp://ftp.htc.honeywell.com/pub/vhdl/token_std-p.vhdl

  ftp://ftp.htc.honeywell.com/pub/vhdl/token_std-b.vhdl

- Generic Distribution Language package:

  ftp://ftp.htc.honeywell.com/pub/vhdl/gdl_pack-p.vhdl

  ftp://ftp.htc.honeywell.com/pub/vhdl/gdl_pack-b.vhdl

- VHDL Strings package:

  gopher://gopher.vhdl.org:70/11/vi/validation/vsp.vhd

  or

  ftp://ftp.vhdl.org/vi/validation/vsp.vhd

- IEEE VHDL math packages:

gopher://gopher.vhdl.org:70/11/vi/math/package

- Random number generator package:

  ftp://erm1.u-strasbg.fr/pub/vhdl/packages.vhdl/random1.vhdl/random.pkg

  other useful math packages, including another random package, reside at:

  http://rassp.scra.org/information/public-vhdl/models/math.html

- E&F VHDL model repository, including all IEEE packages:

  http://rassp.scra.org/information/public-vhdl/models/models.html

- Omniview contact for obtaining the Honeywell Performance Model Library or the Omniview Performance Modeling Workbench

  Jay Runkel - runkel@omnivw.com

7. References

[1] Hein, C., et al, ““RASSP VHDL Modeling Terminology and Taxonomy - Revision 1.0,” *Proceedings 2nd Annual RASSP Conference*, pp 273-281, Arlington, VA, July, 1995.

[2] Aylor, J., et al., Performance and Fault Modeling with VHDL, J. Schoen Editor, Prentice-Hall Inc., pp. 22-144, 1992.

[3] Kumar, S., et. al., *The Codesign of Embedded Systems*. Kluwer Academic Publishers, Boston, copyright 1996.

[4] Pridmore, J., and W. Schaming, “RASSP Methodology Overview,” *Proceedings 1st Annual RASSP Conference*, pp 71-85, Arlington, VA, August, 1994.

[5] Rose, F., T. Steeves, and T. Carpenter, “VHDL Performance Models,” *Proceedings 1st Annual RASSP Conference*, pp 60-70, Arlington, VA, August, 1994.

[6] Steeves, T., et al, “Evaluating Distributed Multiprocessor Designs,” *Proceedings 2nd Annual RASSP Conference*, pp 95-102, Arlington, VA, July, 1995.

[7] Shackleton, J., T.Steeves, “Advanced Multiprocessor System Modeling,” *Proceedings Fall 1995 VIUF*, pp 8.21-8.28, Boston, MA, October, 1995

[8] Meyassed, M., R. McGraw, J. Aylor, R. Klenke, R. Williams, F. Rose, and J. Shackleton “Framework for the Development of Hybrid Models,” *Proceedings 2nd Annual RASSP Conference*, 147-154, Arlington, VA, July, 1995.

[9] Mohanty, “An Integrated Design Environment for Rapid System Prototyping, Performance Modeling and Analysis using VHDL”, Masters Thesis, University of Cincinnati, September, 1994.

[10] Sternheim, E., et al., Digital Design with Verilog HDL, Automata Publishing Company, pp 5-28, 1990.

[11] Zuerndorfer, B., and G. Shaw, “SAR Processing for RASSP Application”, *Proceedings 1st Annual RASSP Conference*, pp 253-268, Arlington, VA, August, 1994.

[12] Hein, C., and D. Nasoff, “VHDL-based Performance Modeling and Virtual Prototyping”, *Proceedings 2nd Annual RASSP Conference*, pp 87-94, Arlington, VA, July, 1995.

[13] Dungan, W., et al., “ Heterogeneous Modeling with ADEPT and PML: Using the PML Lightweight Processor in the Lockheed Martin ATL SAR Model,” UVa Center for Semicustom Integrated Systems Technical Report No. 22903-2442, December 18, 1995

[14] Honeywell Technology Center, “VHDL Hybrid Model Library Style and Methodology Guideline,” Version HML_00.6a, December 22, 1995

[15] Honeywell Technology Center, “VHDL Hybrid Model Library User and Reference Manual” Version HML_00.6a, December 28, 1995