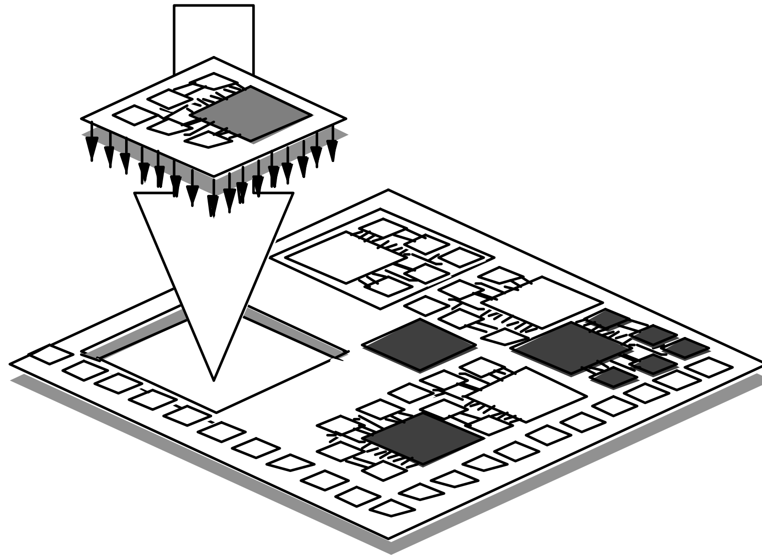
VSI Alliance™
System-Level Interface
Behavioral Documentation Standard
(SLD 1 1.0)

System-Level Design
Development Working Group

Released March 2000

Revision March 24, 2000



Dedication to Public Domain

VSI Alliance hereby dedicates all copyright that VSI Alliance holds in this **______** (the "Work") to the public domain, free of charge, and for the general benefit of the public at large.

VSI Alliance intends this dedication to be an overt act of relinquishment in perpetuity of all present and future rights that VSI Alliance may have in the Work under copyright law, whether vested or contingent, including without limitation, the right to prevent others from freely reproducing, distributing, transmitting, using, modifying, building upon or otherwise exploiting the Work for any purpose, commercial or non-commercial, or in any way.

VSI Alliance understands that such relinquishment includes the relinquishment of all rights to enforce (by lawsuit or otherwise) any copyrights that VSI Alliance may have in the Work.

IMPORTANT - NO WARRANTY. THE WORK IS PROVIDED "AS IS", "WHERE-IS", WITHOUT WARRANTY OF ANY KIND. WITHOUT LIMITING THE GENERALITY OF THE FOREGOING, VSI ALLIANCE EXPRESSLY DISCLAIMS ALL WARRANTIES WITH RESPECT TO THE WORK, WHETHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, WARRANTIES OF TITLE, NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, AND IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

System-Level Interface Behavioral Documentation Standard Development Working Group

Version 1 1.0

Members of the Development Working Group:

Alcatel Microelectronics
AXYS Design Automation
Co-Design Automation
Conexant Systems
ECSI
Fujitsu Microelectronics
ICL High Performance Systems
Improv Systems
LSI Logic
Motorola
Nokia
OKI Electric Industry
SICAN GmbH
Infineon Technologies
Synopsys
ARM
Cadence Design Systems
CoWare
Easics NV
Frontier Design
Hewlett-Packard
IKOS Systems
Integrated Chipware
Mentor Graphics
National Semiconductor
Nortel
Philips Semiconductors
Telefonaktiebolaget LM Ericsson
ST Microelectronics
Toshiba

Active Contributors:
Brian Bailey .... Mentor Graphics
Gjalt de Jong .... Alcatel Microelectronics
Christopher Lennard (Chair) .... Cadence Design Systems
Pete Hardee .... CoWare
Kamal Hashmi .... ICL High Performance Systems
Patrick Schaumont .... Individual Member

Other Contributors
Mark Buckner .... Individual Member
Jean-Paul Calvez .... Individual Member
Larry Cooke .... Cadence Design Systems (Consultant)
Mark Genoe .... Alcatel Microelectronics
Lisa Guerra .... Conexant Systems
Anssi Haverinen .... Nokia
Alberto Sangiovanni-Vincentelli .... Individual Member

Technical Editors
Sybil Sommer
Wilsa Schroers
Herb Leeds

Revision History

Revision	Date	Name	Comments
0.1.0	12/03/98	Christopher K Lennard	Conversion of Proposal into Spec Format
0.2.0	12/28/98	Christopher K Lennard	Inclusion of December 7th Meeting input
0.2.2	1/26/99	Brian Bailey	Contribution to Section: 2.3.2 Attributes
0.2.3	1/27/99	Christopher K Lennard	Incorporation of Gjalt de Jong's comments
0.2.4	2/09/99	Section Leads	Contribution to Sections
0.2.5	2/23/99	Brian Bailey, Gjalt de Jong	Contributions to Sections for Feb Meeting
0.2.6	3/16/99	Section Leads	Content Input (All Sections) – Non-structural edits from Feb Meeting
0.3.0	3/29/99	Christopher K Lennard	Structural Edits from Feb Meeting
0.3.1	4/28/99	Section Leads	Corrections on Version 0.3.0
0.5.0	4/30/99	Christopher K Lennard	Prep for release to SLD DWG review
0.6.0	7/06/99	Christopher K Lennard	Incorporate review comments
0.6.0A	8/17/99	Wilsa Schroers	Edit
0.7	8/23/99	Stan Baker	Format for TC Review
0.7	1/14/00	Herbert D. Leeds, Christopher K. Lennard, Patrick Schaumont	Edit, Insert of Constant Multiplier VC document as Appendix C
1.0	2/00	Wilsa Schroers	Copy edited and formatted document in FM.
1.0	13Mar00	Stan Baker	Edited Figures, inserted license page
1.0	22Mar00	Stan Baker	Edited Figures, formatted for final release
1.0	24Mar00	Stan Baker	Replaced Figure 8, edited tables

Please send comments/questions to:
Interface Subgroup of the VSIA SLD DWG
(vsisldifsg@vsi.org)

Chair: Christopher K. Lennard, Cadence Design Systems
555 River Oaks Parkway, Bldg 4, MS 4B1, San Jose, CA 95134
Phone: (408) 944-7441; Fax: (408) 894-2996; E-mail: clennard@cadence.com

Preface

Purpose of this Document:

The primary objectives of the *System-Level Interface Behavioral Documentation Standard* are:

- **Improved Comprehension:** To develop mechanisms through which VC interface behaviors can be quickly understood at the system-level, enabling rapid but accurate choices to be made.
- **Improved Description:** To provide a mechanism for ensuring the complete description of VC communication at various levels of abstraction.
- **Improved Model Generation:** To enable the clean dissociation of the functional behavior of a block from its communication mechanisms, thereby easing the reuse of the same block with different interfaces (whether synthesizable or "attachable").
- **Improved Integration:** To provide a process for linking together the set of system level VC models at various levels of abstractions of an interface. The linking process must ensure valid property inheritance throughout the abstraction hierarchy.

Who Should Read This Document:

This document is of interest to all involved in the VC production or integration process. More specifically, this document will provide:

- For VC providers a quick and complete way to document existing interface properties of a component, building the VC integrator's confidence in component integratability.
- For VC integrators a simple way to extract higher-level interface operation principles, allowing integration-exploration and test to commence earlier.
- For EDA vendors to help in driving tools and methodologies to implement interface design and IP integration and exchange at the system-level.

Required Background for this Document:

Reading of this document requires:

- Understanding of the importance and usefulness of interface specification as described in the VSIA SLD Interfaces group document, *The Motivation for Interface Based Design*.
- Understanding of the terminology used. The terminology used is that defined in *VSIA System-Level Design Model Taxonomy*, and the extension*, Interfaces: Taxonomy & Terminology*.
- A view towards the importance of not only describing the implementation of a VC interface, but also the importance of hierarchy in linking details of such a description to the operational principles behind any protocol.
- Understanding of the existing common practices used to document VC interfaces. Existing documentation techniques from within a company can be restructured to fit the general format of this standard.

Executive Summary:

This document provides a template to ensure adequate documentation of any VC interface hierarchy. It leaves the actual syntactic representations of any executable model open to the VC provider. However, the document provides a framework by which any such particular representation is related to a standardized high-level syntax. This high-level syntax is described.

The documentation does not require that all interface abstractions described be represented by an executable model. Some interface layers may be included for clarity of description only. The documentation standard provides a structure to describe the:

- Core set of interface layer abstractions which must be delivered with a VC.
- Method for specifying the interface layering principle adopted by the VC provider (flexible).
- Structure for description of each interface layer. Defines a standard behavioral syntax.
- Link between interface and VC implementations or models.
- Association between layers of interface and the maintaining of operational principles.

Table of Contents

1. Overview 1

1.1 Scope 1

1.1.1 Principle of Interface-Based Design 1

1.1.2 Goals 1

1.1.3 Document Organization 2

1.2 Referenced IP 2

1.3 Definition of Basic Terms 2

1.4 System-Level Interface Description Overview 3

1.4.1 Conceptual Motivation 3

1.4.2 Interface Layering Principle 4

1.4.3 Basic Technical Principles 6

1.5 Structure of the SLD Interface Description. 9

1.5.1 Overview of System-Level VC Documentation 12

1.6 Summary of Deliverables 13

1.6.1 Mandatory Interface Layers 15

1.7 How to Read this Document 15

2. Documentation Guidelines 17

2.1 Layering Principle: Section 3.1 17

2.2 Layer Specification: Section 3.2 18

2.3 Data Types: Section 3.2.N.1 18

2.3.1 General Discussion 18

2.3.2 Documentation Requirements. 20

2.4 Internal VC Behavioral Description: Section 3.2.N.2 20

2.4.1 General Discussion 20

2.4.2 Documentation Requirements. 20

2.5 Interface Overview and General Properties: Section 3.2.N.3.1 21

2.5.1 General Discussion 21

2.5.2 Documentation Requirements. 21

2.5.3 Documentation Options 21

2.6 Interface Structure 21

2.6.1 Interface-Specific Description: Section 3.2.N.3.2 $InterfaceName . 22

2.6.2 Structural – Port Identification: Section 3.2.N.3.2 S1 $InterfaceName22

2.6.3 Structural – Inter-Layer Static Mappings: Section 3.2.N.3.2 S2 $InterfaceName. 23

2.7 Interface Behavioral Descriptions 27

2.7.1 Behavioral – Port Attributes: Section 3.2.N.3.2 B1 $InterfaceName 28

2.7.2 Behavioral – Port Transactions: Section 3.2.N.3.2 B2 $InterfaceName. 36

2.7.3 Behavioral – Inter-Layer Behavioral Mapping: Section 3.2.N.3.2 B3 $InterfaceName. 39

2.7.4 Behavioral – Protocol Description: Section 3.2.N.3.2 B4 $InterfaceName 40
2.8 Behavior/Interface Association: Section 3.2.N.4 43
2.8.1 General Comments 43
2.8.2 Documentation Requirements: 43

3. Known Issues 45

4. References 47

5. Appendices 49

A. An Example System-on-Chip Interface Hierarchy 51

B. Layer 1.0 to Layer 0.0 Behavioral Association 55

C. Interface-Layering Documentation Example 57

1. Overview

1.1 Scope

This standard provides a systematic documentation technique for system-level virtual component (VC) interfaces. A VC interface is the information-transfer boundary between a VC internal behavior and any communication channel connecting VC implementations or VC models. System-level VCs are defined as those which exist in any abstraction at or above the cycle-accurate level.

This standard applies to VC interfaces, supporting material, and documentation used by system integrators, virtual component developers, and communication channel developers, when using or developing mechanisms for intercommunication between VCs as defined in this standard.

1.1.1 Principle of Interface-Based Design

The principle of this work is the separation of internal VC behavior that is system-generic, from VC interface protocol that is implementation specific. This separation of behavior and interface is critical to the building of any consistent system-design flow, which moves from architecturally independent functionality to final architecture and fully integrated systems. The separation of VC and interface is also critical to mix-and-match of abstract VC models.

1.1.2 Goals

The purpose of the Virtual Socket Interface Alliance (VSIA) is the encouragement of VC transfer and system-on-chip (SoC) integration. A technique for rigorously and uniformly specifying system-level VC interfaces will improve transfer and integration through:

- **VC Understanding and Utilization**: To reduce the time required for understanding VC behavior correctly. A fast understanding of VC behavior and its relationship to implemented interface protocol allows the system architect to integrate models and explore many more options before committing to the design phase. Furthermore, a complete definition of the interface abstraction hierarchy allows designers, architects, and software authors to work within their preferred area of expertise (for example, embedded-software, RTL, and so forth). They are still able to effectively communicate between the different design abstractions (for example, unified test-benches/test-results can be applied to any view of the design).
- **VC Model Supply and Generation**: To assist in the building of VC models at a level higher than RTL that is not specific or specialized for a single use. First, an intelligent common approach to the abstraction of interface operation will help in quickly identifying generally useful abstract models. Thus, every IP consumer will not require a *different* abstract model. Second, a common interface language or capability will allow portability of models between tools. This means that models do not have to be built separately for each simulation environment.
- **VC Integration Speed**: To assist in moving more quickly to the final integration process by providing a clean mechanism for mapping abstract system communication into physical (hardware or software) implementations. The abstractions will also provide a simple mechanism for translating the test-benches used for early-system validation into integration test-benches.
- **VC Protection**: That is a side effect of capturing the complete interface specification for a VC. Supplying the interface behavior with a high level algorithmic behavior, if needed, allows a user to verify the functional or performance fit of a VC into a system without revealing implementation specifics.

To provide a solution that addresses all of the elements described above, the primary objectives of the interface group are:

- **Improved Comprehension:** To develop mechanisms through which VC interface behaviors can be quickly understood at the system level, enabling rapid but accurate choices to be made.
- **Improved Description:** To provide a mechanism for the complete description of interblock communication provided at various levels of abstraction above RTL.

- **Improved Model Generation:** To enable the clean dissociation of the functionality and behavior of a block from its communication mechanisms, thereby easing the reuse of the same block with different interfaces.
- **Improved Integration:** To provide a process for linking together the set of abstract models at various levels of abstractions of an interface. The linking process must ensure valid property inheritance throughout the abstraction hierarchy.

1.1.3 Document Organization

This document contains the following sections:

- Section 1 provides an overview including background, a summary of contents, and required deliverables for compatibility with this standard.
- Section 2 provides a set of specific VC system-level interface documentation guidelines.
- Section 3 contains known issues opened by this standard to be resolved in future versions.
- Section 4 contains a set of references relevant to the development and motivation for this standard.
- The appendices give several examples illustrating the application of the *System-Level Interface Behavioral Documentation Standard*.

1.2 Referenced IP

There are none at this time.

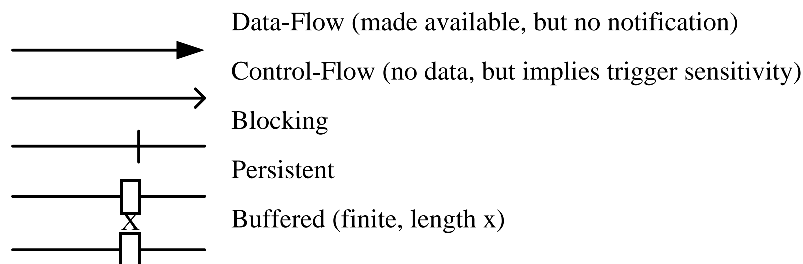
1.3 Definition of Basic Terms

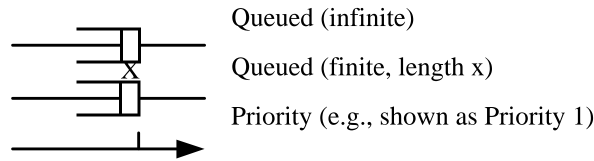
Any discussion on the topic of the system-level VC interface must possess a clear definition of the term "interface" and associated properties. This definition must be sufficiently clear to develop a clean separation between the concept of component behavior and component interface. Furthermore, any of the assumed but heavily overloaded common usage terms associated with interfaces and communications must be disregarded in favor of a clear and comprehensive terminology set for system-level design interfaces.

The terminology adopted by this standard is described in the *Interfaces: Taxonomy & Terminology* document. Reading of the *System-Level Interface Behavioral Documentation Standard* requires knowledge of the correct use of terminology as defined in the context of system-level design.

The *Interfaces: Taxonomy & Terminology* document must be examined prior to reading this work.

In addition to the terminology, there is a basic visual syntax provided with this standard. This syntax is used for the description of communication behaviors at an abstract level. The basic set of elements in this syntax is:





Some of these behaviors are additive, for example blocking and priority. The attributes and valid combinations of attributes are described further in Section 2.7.1, Behavioral-Port Attributes. (ASCII representations for these elements are provided in Section 2.7.1.2.)

1.4 System-Level Interface Description Overview

The following sections provide a description of the system-level interface.

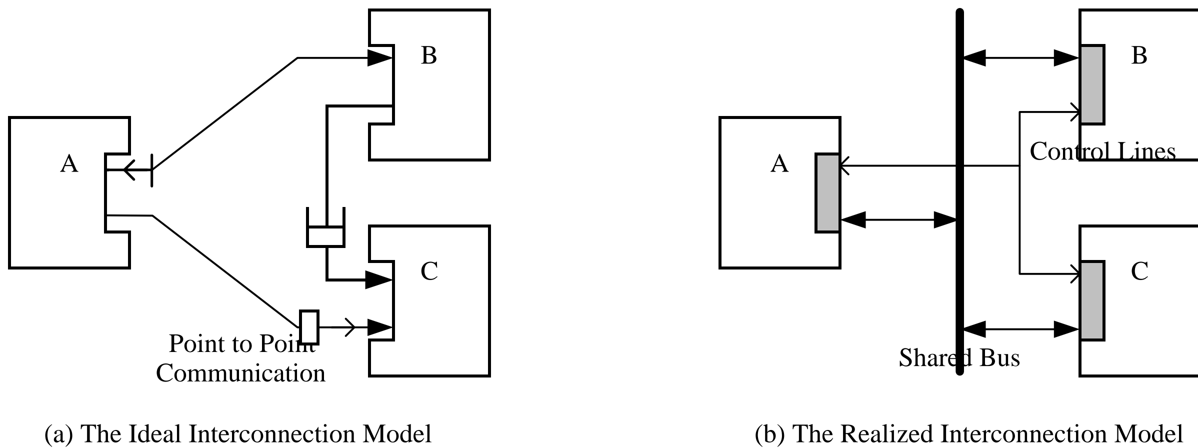
1.4.1 Conceptual Motivation

The interface-based design [2] is explained in an example design flow.

Commonly, system architects begin considering their problem from the perspective of functional or behavioral task definition. These tasks are linked by “ideal” channels, through which information is sent and received as needed, without concern for any conflicting resource requests. There is no need for such concern at this stage, as the architecture is dealing with functionality and not communication protocols.

As the design is refined, common communication resources are specified, control protocols administered, and sharing of functional units identified. The common issues associated with system design become visible, and the design moves from that of the ideal, Figure 1(a), to that of the realizable, Figure 1(b).

Although at the finest level of detail, the design may appear to have changed relative to high-level conceptual models, the fundamental operational principles must be inherited. The implemented or “realized” tasks of Figure 1(b) must perform the conceptual tasks of Figure 1(a). In fact, the model of Figure 1(a) should remain completely valid throughout the design refinement process. This refinement can be viewed as one of tightening the design envelope until the design space converges to an implementation. The implementation involves greater complexity and dependence upon the actual channels of communication used to transfer information within the design.



(a) The Ideal Interconnection Model

(b) The Realized Interconnection Model

Figure 1: Conceptual Layering of Connectivity Models

The *System-Level Interface Behavioral Documentation Standard* introduces the concept of interface layering, or abstraction hierarchy. An interface layer is a translation wrapper that is capable of taking one level of interface abstraction to the next more detailed level. For example, if a VC behavior uses the conceptual communication channel of a blocking read at the top-most abstraction layer, one layer of refinement below this can define the blocking read as a read action dependent upon an asynchronous Request/Acknowledge action. A further layer below this, the Request/Acknowledge actions could be tied to synchronous clocking actions in a final hardware implementation. In each case, it is critical to understand:

- What the conceptual reasoning is for an interface action (top down justification).
- How the conceptual action is implemented in the final design (bottom up verification).

This understanding can be established by linking the interface layers, or abstractions, through a clear refinement mechanism.

Abstraction layers of an interface may or may not correspond directly to implemented interfaces. As a particular example of an implemented interface abstraction hierarchy, consider the layering concept in the writing of embedded software. An example of the layering concept is the writing of embedded software for a system-on-chip. The greatest parallelism in the process of design refinement is achieved if the software and hardware elements are developed largely independently. The software author assumes that the software routines are "notified" of the necessity to perform a task and the presence of relevant data. The hardware designer assumes that the software will initiate transfers as required and correctly interpret the exercising of the interface protocols. A clear hardware/software interface specification is the solution to this concurrent-development problem. The hardware to software developments are integrated into the SoC through software to hardware interface-protocol code elements known as *hardware driver*s. Also useful would be the specification of a standardized transaction set with which software modules can interact with these drivers.

The issue faced in system-level design is a generalization of the hardware/software co-specification and co-design problem described above. The development of any smooth, consistent flow from the system level to implementation requires that behaviorally critical communication properties are verifiably inherited and implemented. This requirement is dominated by the need to specify, in a standard way, multi-level VC interfaces, which are complete-by-construction and verifiably correct across the multiple levels of abstraction. Meeting this need ensures that the implementation will correctly fulfill the system specification requirements.

1.4.2 Interface Layering Principle

To introduce the concept of multi-level VC interfaces and what this implies upon design, consider the diagrams in Figure 2. The four options presented (a, b, c, d) represent possible VC implementation and documentation styles compatible with this standard. These diagrams show how the hierarchy of interface layering introduces an "onion-like" view of a VC. This "onion" moves the VC integrator's understanding from the channel-generic internal behavioral description to the specifics of implemented protocol. To understand how these layered interface views fit within the system-design refinement flow, refer to Appendix A.

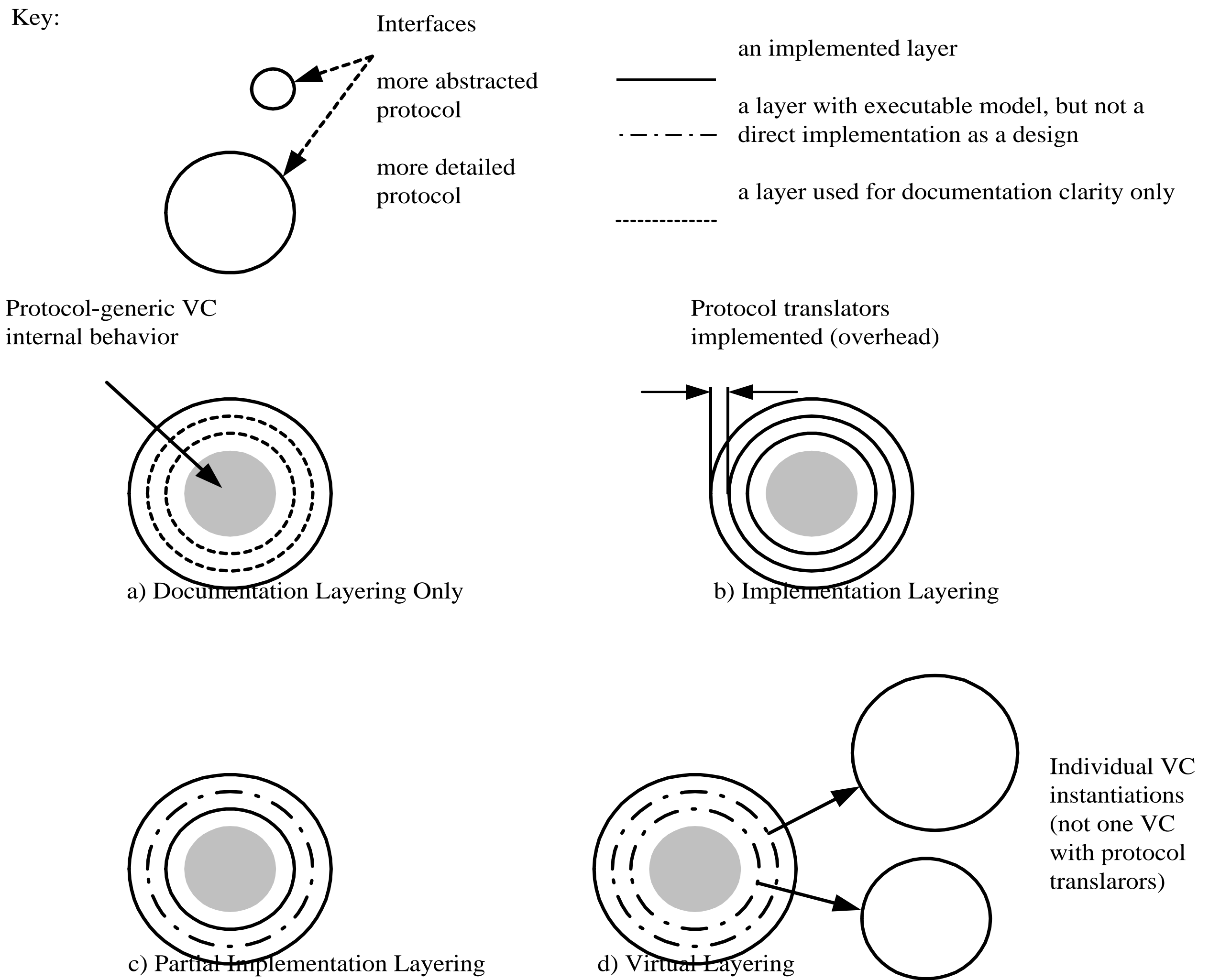


Figure 2: Layering Principle for VC Interfaces

Each layer of a protocol interface hierarchy can exist in several forms:

- *Documentation* – implying that the interface behavior is completely described and related to the abstraction-layers above and below it.
- *Implementation* – implying that the VC exists for integration into the SoC with this level of interface abstraction exposed.
- *Executable* – implying that a VC model exists with this interface abstraction exposed to allow integration of the VC model into the SoC design/simulation environment with abstract communication channels.

Note: An *Executable* implies only sufficient model detail for simulation purposes during system analysis. The existence of an Executable model for an interface layer does not imply that the actual VC implementation can be physically stripped back to this set of abstract communication properties.

Descriptions of the four different interface-layering principles for VC delivery (Figure 2) follow:

- **Documentation Layering Only:** Only the "skin of the onion" – the most detailed level of protocol definition – corresponds directly to an implementation. All other interface layers are just documentation provided to help guide the VC integrator's understanding from the high-level VC behavior (inner kernel) to the protocol-specific "skin" of the onion.
- **Implementation Layering:** Every layer of the VC onion corresponds to a physical implementation and should be accompanied with adequate documentation and executable models. Each progressively more refined interface layer is a protocol "enhancement," taking the design from one that is system implementation generic to a component with a coupled protocol block very specific to a system configuration. If the system integration process can support more generic VC protocols (e.g., through guided synthesis of interface blocks), the more "partially-peeled" onion can be used. Like use of the OCB VCI [1] for translation from bus transactions to another specific system bus protocol, this implementation layering approach implies the design overhead of physical translation from one level of interface protocol abstraction to another.
- **Partial Implementation Layering:** Some interface protocol abstractions are implemented, some are just provided in model executable form for simulation or verification purposes, and others may solely consist of documentation to conceptually link the layers. The translation between the implemented protocol abstractions may imply design overhead.
- **Virtual Layering:** Looks like implementation or partial implementation layering, but without the overhead. The actual VC implementation does not have a set of protocol translators from more generic to more system specific protocols. Instead, each conceptual "layer" is actually a new VC with a fully integrated protocol block. That is, for a set of related designs, the single VC documentation is of all the same internal behavior, with each specific implementation tuned to support the interface abstraction the VC Integrator chooses to use. Hence, a three-layer implementation implies three VCs, but conceptually it makes sense to think of these layers as one "layered" object.

Note that the commonly recommended "onion" is partial implementation layering. This layering option is one in which the inner layers are: documented, have executable models (and layer translators), and have transition to outer layers which are implemented. There may only be a few implemented inner layers to keep overhead low.

This standard supports each of the above "multi-level" interface description techniques for VC delivery. Use of the standard does *not* directly imply an increase in design overhead. The degree of implementation associated with any layer is left to the discretion of the VC provider. Note that in this documentation standard we are mandating only that a VC be supplied with (at a minimum):

- *A detailed protocol level description* (Layer 0.0). This corresponds to an implementation of the component. Details of these requirements are found in the VSIA Architecture document.
- *A description of the interface to the internal behavior* (a Layer 1.0 interface model). This is the interface to the "core" behavior of the VC, which is specified independent of system implementation assumptions. This layer is mandated for reasons of documentation clarity even if an executable model does not exist at that level of abstraction.

Current trends in the industry suggest that certain abstract communication behaviors may be directly synthesizable, while others are mainly for simulation and verification purposes. However, this concept of the "synthesizable subset" of behaviors is not fully defined in the industry, and hence is not specifically represented within this standard. However, there are a set of proprietary standards in existence and some other general standards emerging (e.g., OCB VCI [1]). General usage of these standards in the future may permit hand-off of VCs with intermediate (Layer 0.x) interface descriptions.

It is important to note that in the examples in Section 1.4.3, we are detailing this "onion" concept by extending from the interface properties necessary for the bulk of VCs transferred today. It may not be necessary to specify all of the components shown in Figure 3, such as control and protocol blocks and their refinements shown in Figure 4 and Figure 5, for all VCs. In fact, we strongly recommend that the interfaces for simpler (peripheral) VCs be kept as simple as possible, in order to facilitate easiest reuse.

1.4.3 Basic Technical Principles

We may think of an interface to a VC as the set of components depicted in Figure 3. Associated with the operation of an interface into the external system (other VCs) are two aspects of control:

- The requests for transfer of information to and from the behavioral block.
- The control of the block associated with a specific protocol (protocol block).

The former of these encompasses the basic communication concepts such as *Read* or *Write* data. The latter is associated with such operations as handling a request-acknowledge sequence, managing a split-transaction on a bus, etc. The protocol block is the unit of functionality that translates abstract notions such as, *Write_To_VC_Alpha,* into concrete actions such as, *Toggle_Line_CS*.

As depicted in Figure 3, the protocol block has no hierarchy. It is just a simple, flat translator. The behavioral interface shown is the Layer 1.0 Interface for the VC, and the component interface is the Layer 0.0 Interface for the behavioral block coupled with the interface protocol.

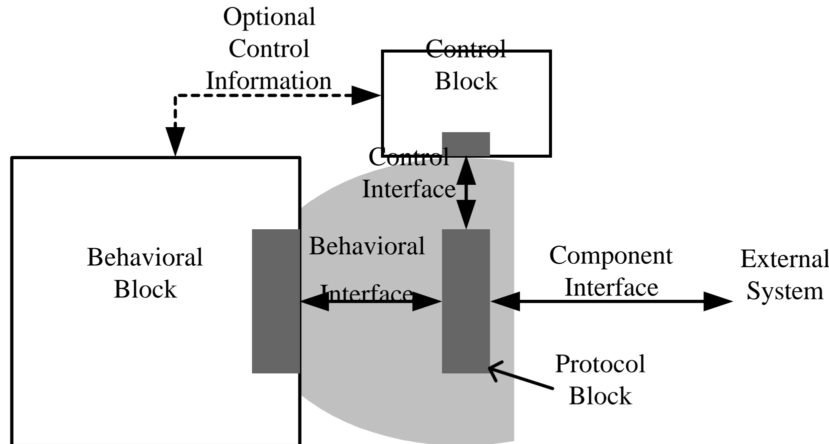


Figure 3: Interface Model for System-Level VC

Although the flat protocol block is a conceptually straightforward model, it is necessary that we support the concept of interface hierarchy. To that end, we can lump that which is referred to above as the protocol control block in with the protocol block itself, generating the much simpler interface model of Figure 4. This concept is the generalization of the virtual component interface (VCI) definition of the VSIA OCB Standard [1]. That is, the OCB VCI exists as one layer in a possible interface-layer abstraction hierarchy (i.e., an interface hierarchy that resolves to bus communication). The hierarchical system-level interface described in this documentation standard is referred to as the *System-Level VCI* (SL-VCI).

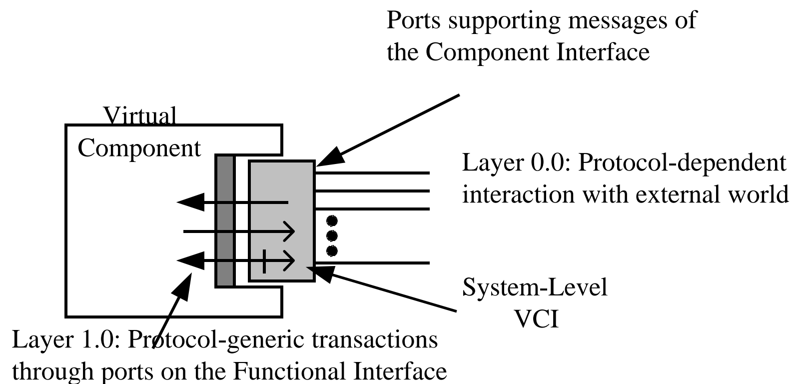


Figure 4: Interface Specification – Layers 0.0 and 1.0

Transactions through the behavioral interface to the SL-VCI may now be specified and associated with a set of attributes, which describe critical properties about the inter-relationship between the VC behavior and the principles of the expected protocol operation. For example, it may be necessary for the correct operation of a VC that blocking-read be supported. If the SL-VCI attached to this virtual component has neither the capability to inform the VC blocks producing input data that the blocking-read action has taken place, nor the ability to support the queuing responsibilities associated with not informing the VCs producing input data, then the system will fail. Techniques for system communication description similar to that described already exist [6], but until now these techniques had not been standardized.

The definition of the required transactions and the attributes that must be supported through the behavioral interface provide a very well defined way of assessing the applicability of the system protocols being considered for the VC.

The SL-VCI may now be unraveled through a series of interface abstractions at successively lower levels of abstraction (each equivalent to the next "onion skin" outward). Each of these abstractions must be complete in that it must implement all of the transaction behaviors required of the abstraction layer above it. The unraveling of the SL-VCI identifies both more refined protocol blocks as well as protocol control blocks. These blocks may be regarded as VCs (such as, a protocol control block might contain an RTOS) to which the same abstract interface layering principles may apply. This concept is shown in Figure 5.

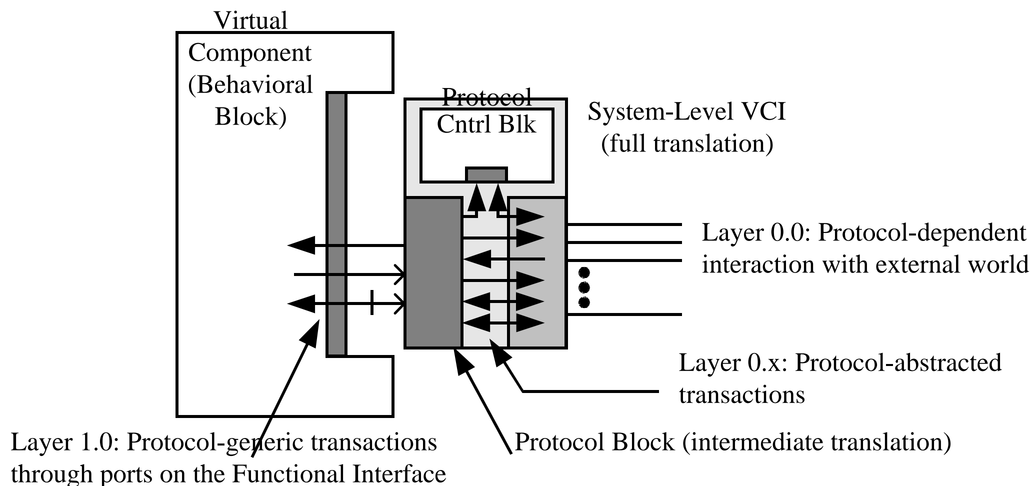


Figure 5: Interface Specification – the Intermediate Layers

The completeness of each abstraction layer in the VC interface-block permits the valid connectivity of components at each of these levels. It is the proper hierarchical structure of these communication principles that will move the industry toward a consistent system design refinement process.

Within the *System-Level Interface Behavioral Documentation Standard* a transaction set is defined, which is sufficient to describe the protocol behaviors that pass across an interface, regardless of the level of abstraction. At the higher levels, ports and transactions through those ports might be assigned any of a number of well-defined communication control principles (attributes). At the most refined level of protocol definition, these attributes will be simple. We are providing a documentation/syntactic-mechanism by which completeness of each abstraction layer is guaranteed. Any set of abstractions compatible with that linking of interface hierarchy will be supported. This includes standards development such as that from the On-Chip Bus DWG of the VSIA [1].

A simple example of the linking of abstract interface transactions (e.g., Layer 1.0) to detailed protocol (e.g., Layer 0.0) is given in Appendix B.

It is not the role of this standard to define the most generally applicable interface abstractions for various system models during the design refinement process. The set of abstractions is to be determined either by the model providers themselves at the time of model development (migrated to a default standard within the industry), or generated as a separate standard by another body. The definition of applicable interface abstractions is not the role of this standard for two reasons:

- The *System-Level Interface Behavioral Description Standard* is not restricted to a particular domain of design or of methodology styles. The standard is more generally applicable in that it just ensures that any such abstraction adopted is complete both in description and in ability to capture communication functionality.
- The VSIA hierarchy of interface abstractions is an extremely valuable conceptual tool. The syntax finally developed could be used to describe an executable for a hierarchy of interface abstractions. However, implementation of such a hierarchy is unlikely to ever be as efficient as various application-specific abstraction-domain simulations for which specialized processing engines can be designed.

1.5 Structure of the SLD Interface Description

Figure 6 shows the flow of the documentation associated with a full virtual component system-level description (including both the interface description as well as the behavioral model description). This standard provides guidelines for Section 3, Technical Attributes, of the *System-Level Virtual Component Interface Documentation*. Following Figure 6 is the detailed structuring of this interface portion of the VC documentation, Section 1.5.1, Overview of System-Level VC Documentation sections. This structure is based upon the general structuring principles described below. In this overview, the document section numbering scheme is introduced. This scheme is both efficient, and compatible with standard document-processing software. The numbering scheme is further detailed in Section 1.7, How to Read this Document.

The following are stipulations for description of system-level interface properties of a VC:

- It is mandatory that all system-level VCs be accompanied by at least a black-box behavioral model, which describes the internals of the VC, and communicate with this behavioral model described using Layer 1.0 Interface principles. If a VC is offered with a protocol block integrated into the design, the VC documentation must also include a full translation from the set of abstract Layer 1.0 interface transactions to the Layer 0.0 interface messages. This translation may be through a series of Layer 0.x interface abstractions. It is not mandatory that behavioral models corresponding to each Layer 0.x abstraction be provided.
- A VC may have multiple interfaces at any specific abstraction layer. Interface documentation proceeds in a layered approach describing all interfaces at a specific layer prior to progressing onto the next level of protocol refinement. Specific interface documentation can be identified by the name of the interface within each layer. (For example, *Section 3.2.N.3* contains all interface descriptions for the abstraction level corresponding to increment N.) Separate interfaces to a VC must be defined as independent entities. That is, at Layer 1.0 there can be no explicit timing/sequencing constraints between separately defined interfaces. Implicit relationships between Layer 1.0 interfaces to a VC must be managed by the behavior of the VC itself.
- The documentation structure for a system level interface must be categorized using the following format. (Note Figure 6 and the Overview of System-Level VC Documentation sections.)

  Data-Type Description: Documentation Section ID - 3.2.N.1

  Includes data-types as used in the behavioral model and interfaces. Base types inherited by all abstraction layers are specified in the mandatory description of Layer 1.0 (Section 3.2.0.1). Specifics of data-format (including additional information coupled with a datum such as dataTag, dataPriority, and so forth, are associated with descriptions of the interface structure or interface behavior)

  Overview and General Properties for Interface Abstraction: Documentation Section ID - 3.2.N.3.1

  Assumed behaviors are applied to all ports. May follow as a consequence of the computation domain [5] in which a model exists. Behaviors are described as per the techniques used in single-port behavioral typing (description follows).

  Interface-Specific Descriptions: Documentation Section ID – 3.2.N.3.2

  Consists of a structural description section and a behavioral description section.

  Structural Description:
  Contains the description of ports of the VC interface to which a channel is connected.

  Single-Port Structural Typing (S 1. Port Identification)

  Name, direction, data-type and data-format associated with a port.

  Multi-Port Structural Description (S 2. Inter-Layer Static Mappings)

  Association of a set of ports at this interface abstraction layer to each port at the one-higher interface abstraction layer. A set of ports at this layer may implement the behavior of a single, more abstract port or may break down a more complex data-type into component parts. Association of ports across abstraction layers may be one-to-many, or many-to-one (sharing). Many-to-many is possible but strongly discouraged.

Behavioral Description:

Contains a description of the interface behavior supported by a port, sets of ports comprising an interface, or common across all ports at this abstraction layer.

Single-Port Behavioral Typing (B 1. Port Attributes, B 2. Port Transactions)

Behavioral typing of a port is identified through a standardized set of transactions/messages and behavioral attributes associated with each port. Data-format associated with particular transactions/ messages are also identified.

Multi-Port Behavioral Description

(B 3. Inter-Layer Behavioral Mapping, B 4. Protocol Description)

Identifies relationships among port behaviors at this abstraction layer. This will include behaviors particular to this abstraction, key behaviors that implement the transactions, and attributes from more abstract interface layers. Structuring of this section must be compatible with the key associations delineated in Structural Description *Inter-Layer Static Mappings*. This structuring will further call out dynamic associations as per *B 3. Inter-Layer Behavioral Mapping* of the system-level VC dcumentation sections.

- Links between an interface and the internal variables used within a behavioral model must be defined. These follow the description of the model and its interfaces, which are described independently.

Figure 6 shows the system-level documentation flow.

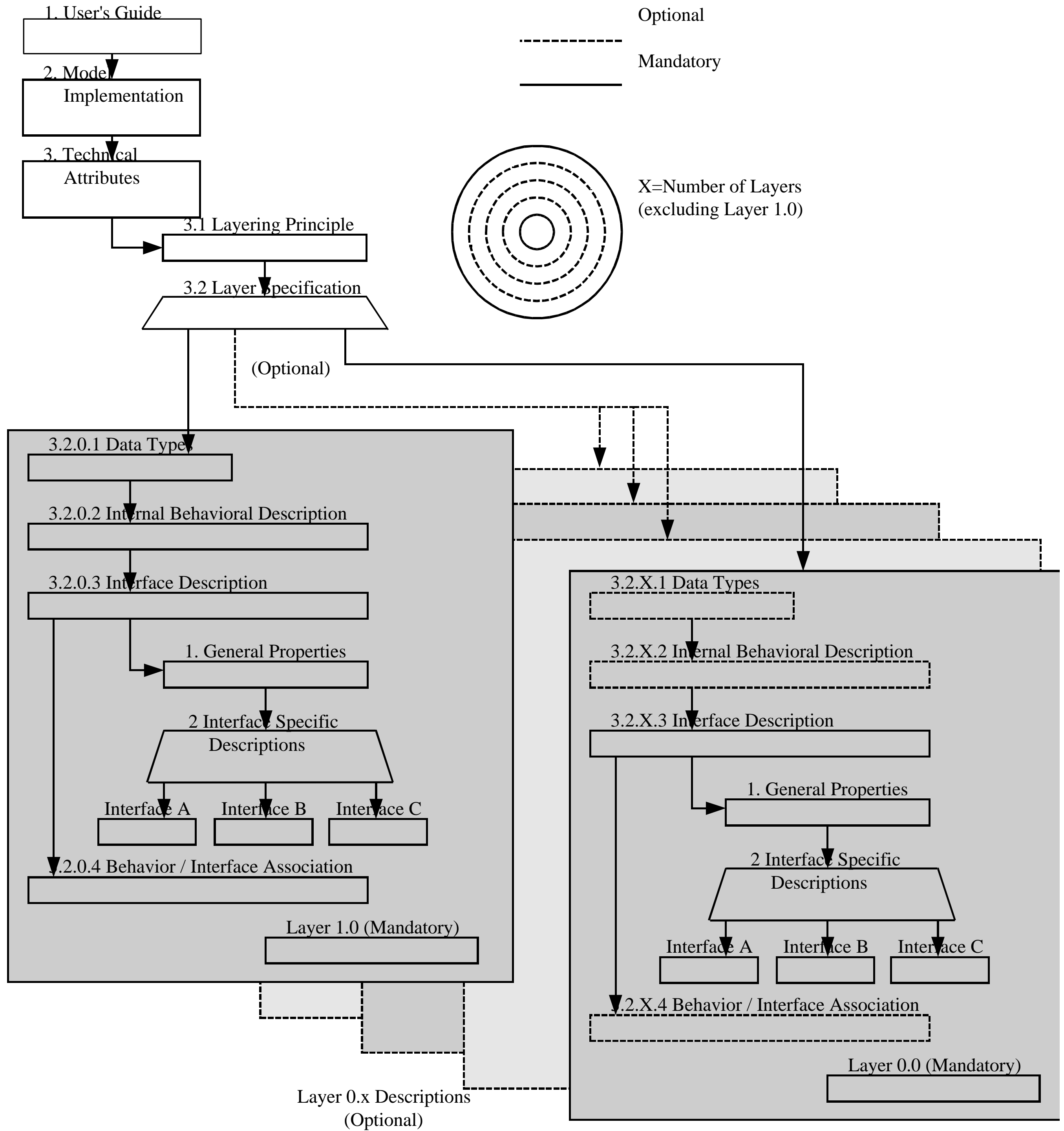


Figure 6: System-Level VC Documentation Flow

1.5.1 Overview of System-Level VC Documentation

1. User's Guide (To be detailed in a full System-Level VC Description Standard)

2. Model Implementation (To be detailed in a full System-Level VC Description Standard)

3. Technical Attributes

3.1 Layering Principle

Describes structuring of the VC abstraction layers in terms of:

a) Number of abstraction layers

b) Name of abstraction layers

c) Provision with each layer (implementation, model, or document only).

3.2 Layer Specification

Note: In section headings, "N" is incremented for each more refined interface layer.

3.2.N.1 (Layer_Name) Data-Types

3.2.N.1.1. General Types

Types used within the layer. May cross-reference the types provided for other layers.

3.2.N.1.2. Transport Types

If desirable, types specific to transport of data at this layer may be broken out.

3.2.N.2 (Layer_Name) Internal VC Behavioral Description

Note: If one model connects at this layer to several interface layers, it must cross-reference to each of these interface layers.

3.2.N.2.1. General Properties

List of general operating assumptions applied to all actions in the operational description. For example, may specify the handling of fixed-point representation.

3.2.N.2.2. Operational Description

Will include the functional and timing properties needed for the adequate integration of the component. In some cases, this may be a black box model with internal details hidden for IP protection purposes. However, under all circumstances, any behavior that influences the proper execution of the interface must be revealed.

3.2.N.3 (Layer_Name) Interface Description

3.2.N.3.1. Overview and General Properties

Identifies the names of all interfaces used at this level. Describes behaviors common to all ports at this layer. The behavioral properties of a domain of computation must be specified here if this interface layer assumes the properties of that domain.

3.2.N.3.2. Interface-Specific Descriptions

Proceeds in a vertical slice approach, describing all structure and behavior associated with each interface before describing aspects of another.

$InterfaceName:

S $InterfaceName: Structural

1. Port Identification

Includes the port names, port class, and port type.

2. Inter-Layer Static Mappings

Associated with hierarchy of data-types, and control across abstraction layers.

B $InterfaceName: Behavioral

1. Port Attributes

Behavioral attributes associated with each port.

2. Port Transactions

Transaction types associated with each port.

3. Inter-Layer Behavioral Mapping

Classifies the port behaviors (transactions and attributes) at the current interface layer by relating them to the set of port behaviors at a more abstract interface layer. Ties the abstract communication actions to the specific protocol elements that execute those actions.

4. Protocol Description

Provided for each protocol implemented at this layer. Structuring of the information follows from the associations called out in the static and the behavioral mappings.

3.2.N.4 (Layer_Name) Behavior / Interface Association
Association between variables in the internal behavioral description and the interface description. Permits integration of executable models or VC implementations.

1.6 Summary of Deliverables

The deliverable requirements, for the *System-Level Interface Behavioral Documentation*, are described in Table 1 and Table 2. Table 1 describes the specification of layers of interface description in terms of the IP block design status (hard, firm, soft). Table 2 describes the set of data, which must be provided with each interface-abstraction layer.

Table 1: Interface-Layer Deliverables with each VC

Interface Abstraction Layer	IP Block Design Status			Commonly Used Formats	VSIA Format(s)
	Soft	Firm	Hard		
Layer 1.0	M	M	M	Document, C, C++, SDL	Document
Layer 0.x	R(1)	R(1)	R(1)	Document, C, C++, SDL	Document
Layer 0.0	M(2)	M (2)	M	Verilog, VHDL, Document	Document

Table 2: Documentation Deliverable with each Interface Layer Description

Section 3.2 Layer Specification	Deliverable	Commonly Used Formats	VSIA Format(s) *	Layer 1.0	Layer 0.x	Layer 0.0
3.2.N.1	Data Types	See Table 1.	document	M	CM(3)	CM(3)
3.2.N.1.1	General Types		document	M	CM(3)	CM(3)
3.2.N.1.2	Transport Types		document	R(4)	R(4)	R(4)
3.2.N.2	Internal Behavioral Desc.		document	M	R	R
3.2.N.2.1	General Properties		document	M	R	R
3.2.N.2.2	Operational Description		document	M(5)	R	R
3.2.N.3	Interface Description		document	M	R	M
3.2.N.3.1	General Properties		document	M	R(6)	CM(6)
3.2.N.3.2	Interface Description		document	M(7)	R	M
3.2.N.4	Behavior/ Interface Assoc.		document	M	R(8)	CM(8)

"(*) Documentation is essential, but executable models may be provided wherever possible. For example, general data types may contain actual compilable templates along with explanatory text."

Explanation of Requirement Assignments:

The following list corresponds to the superscripts in Table 1 and Table 2.

(1) The provision of Layer 0.x descriptions with any VC interface are Recommended. These intermediate-protocol abstractions might be used for improving simulation speed during architectural exploration, or as a refinement mechanism to better understand the Layer 0 specification in terms of the abstract functional interface description, Layer 1.0. However, the abstractions are not Mandatory to ensure that a VC interface behavior is understandable or integratable.

(2) The provision of Layer 0.0 descriptions with any firm or soft VC is Mandatory. Even if the interface to the VC is fairly protocol-generic (allowing integrated synthesis with a more detailed protocol block), the Layer 0.0 model that corresponds to the most detailed level of implementation must be provided.

(3) If data-types used at this abstraction layer are not defined in abstraction layer 1.0, then these types must be defined in this section.

(4) If certain of the defined data-types are only used in the transport of the data rather than the internal operation of the VC, it is Recommended that they be identified separately.

(5) A black box description of the behavior or better is Mandatory.

(6) Mandatory if for the provided interface layer there is a behavior or set of behaviors assumed to apply to all ports or transactions.

(7) Not Mandatory for the Layer 1.0 Interface Descriptions are the: *InterfaceName S1* Inter-Layer Static Mappings, *InterfaceName B3* Inter-Layer Behavioral Mapping and *InterfaceName B4* Protocol Description. The Inter-Layer mappings are not defined, as there is no abstraction-layer above Layer 1.0. Furthermore, the Layer 1.0 System Connectivity Attributes do not allow any protocol implementation to be described at this layer. Any timing or sequencing of transactions at the Layer 1.0 interface is, by definition, part of the functionality/behavior of the VC, and not explicitly of the interface itself.

(8) The specification of the behavior/lnterface association at any layer other than Layer 1.0 is Mandatory if a behavioral model is specifically described for that layer. Such a model must be provided for Layer 1.0, as it is Mandatory.

1.6.1 Mandatory Interface Layers

Layer 1.0 Interface Model

The Layer 1.0 interface model describes the communication requirements of a VC at the highest level of abstraction. At this level (as it may be at other 0.x abstractions above the 0.0 layer), it is an unmapped interface model. That is, this model describes the interface behavior of a VC prior to any consideration of mapping it to hardware and software. The overall communication intent of the VC must be made clear at this level of abstraction.

The interface model may be considered from two different perspectives depending upon the actual VC being considered:

- **VC is a behavior** (that which is to be mapped): The Layer 1.0 interface model specifies a set of high-level communication requirements (protocol-block and/or external environment must supply me with …) and properties (I will supply the protocol-block and/or external environment with the following behaviors:). These ensure that the VC will behave as it was designed.
- **VC is a programmable block** (that onto which a behavior may be mapped): If the VC is a programmable component (for example, a CPU) without inherent behaviors, then the Layer 1.0 model for the component is a high-level description of *enabling* properties (I can pass the following communication properties through my interface:)

Figure 7 illustrates this concept using a simple example.

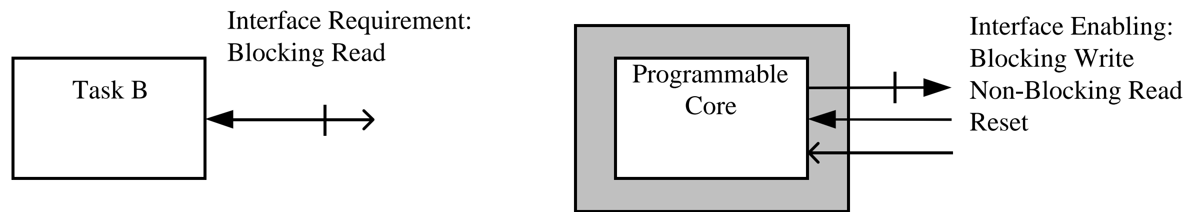


Figure 7: Layer 1.0 for Behavioral and Architectural VCs

In this case, task B cannot be mapped directly onto the programmable core as the fundamental property of blocking read does not cross that boundary.

Layer 0.0 Interface Model

The Layer 0.0 interface model is a fully mapped interface model. It gives interface properties at the RTL-equivalent level for both hardware and software components. In the case of software, this implies, at a minimum, the specification of a symbolic link to a memory map, but may be as detailed as the actual specification of registers associated with the transfer of control and data. Given the current state of the art in transfer of VCs, all components must be specified with this Layer 0.0 link to implementation. In the future, it may become possible to hand-off a component with a more "behavioral" interface description.

1.7 How to Read this Document

Section 2, Documentation Guidelines, describes, in detail, the form of the deliverables of a system-level VC interface. The sections are titled with the names of the sections required in the delivered VC documentation and the section numbers to be shown in that documentation.

The following format has been adopted:

2.$Number $NameOfDocumentSection: Section $SectionIdentifier

where:

2.$Number is a reference to a section in this document.
$NameOfDocumentSection is the section name to be used in the VC interface documentation
$SectionIdentifier is the number/identifier to be used in the VC interface documentation.

For example, in this document, there is a section titled:

2.6.2 Structural - Port Identification: Section 3.2.N.3.2 S1 $InterfaceName

which follows the format described above. The number, N, is assigned by this document formatting. As the document progresses from the highest level of abstraction (Layer 1.0 where N=0) to the most refined level of abstraction (Layer 0.0), this number, N, is incremented. Each N corresponds to a different interface abstraction layer. A specific example follows.

Consider the documentation of a VC with four interface layers assigned the hierarchy numbers: 1.0, 0.6, 0.4 and 0.0. Layer 1.0 will have section numbering N = 0, Layer 0.6 will have section numbering N = 1, Layer 0.4 will have section numbering N = 2, and Layer 0.0 will have section numbering N = 3. So, for Layer 0.4 and an interface with name: DataOut, there will be a section in the VC interface documentation titled 3.2.2.3.2 S1 DataOut: Structural – Port Identification

2. Documentation Guidelines

This section provides VC providers with a detailed template with which to document their VC interface as per the *System-Level Interface Behavioral Documentation Standard*. It also provides sufficient detail about the terms and the structure used to allow the VC integrator to easily comprehend the interface documentation.

Note that all system-level VCs must be provided with a Layer 1.0 interface description, as well as a description of their interfaces corresponding to an implementation of the VC. Any additional description accompanying the documentation, which deals with intermediate abstractions of the *same* VC protocol-block are known as Layer 0.x models. The VC provider is at liberty to assign a particular number to `x' to indicate the relative level of abstraction. The number of intermediate layers is arbitrary.

Not described in this documentation are:

- The *User's Guide* section. (To be described in a more general system-level VC description standard.)
- The *Model Implementation* section. (To be described in a more general system-level VC description standard.)
- Types of Layer 0.x models which might be appropriate for various interface types.
- The specific functional property-set required of any Layer 0.x model (layers other than Layer 1.0 and Layer 0.0).

Where appropriate, the sections of this document are broken into two parts:

- General Discussion: Material suited for introduction to the concept
- Documentation Requirements: How to generate this section of VC documentation (given points raised in the general discussion, if necessary).

When the concepts are simple, the descriptions accompanying sections contain only information on the requirements.

2.1 Layering Principle: Section 3.1

In Section 3.1 (Layering Principle) of the system-level VC documentation, the VC provider must give an introduction to the VC and its layers of abstraction. This introduction is to describe the general structure of the model and documentation layering. This structure is to be followed in Section 3.2: Layer Specification of this VC documentation. An example technique for creating this representation is the "VC onion" as described in Section 1.4.2. At a minimum, this introduction to the layering principle must:

- List the Names (and Layer ID as a number between Layers 0.0 and 1.0) for all layers.
- Provide a note as to whether each layer is accompanied with:
  - Documentation of the layer without the executable model
  - An executable model corresponding to the behavioral model at that layer
  - An executable model corresponding to the interfaces at that layer
  - An implementation directly implementing the abstraction of that interface layer

This basic structure should be given in the tabular form as shown in Table 3.

Table 3: Structure of the Interface Layering Principle

Layer ID	Layer Name	Document Section	Behavioral Executable Model	Interface Executable Model	Implementation
Layer 1.0	Name_1	3.2.0	Y	Y or N	Y or N
…	…	…	…	…	…
Layer 0.x	Name_X	3.2.[*1]	Y or N	Y or N	Y or N
…	…	…	…	…	…
Layer 0.0	Name_0	3.2.[*2]	Y	Y or N	Y

Superscripts (*1), (*2): As per Section 1.7 on the heading structure for the VC documentation, the number in the 3.2.N. field is given by the progression of document section numbers. For example, if Layer 0.x is one level of abstraction below Layer 1.0, then the Document Section would be 3.2.1.

Note: It is optional to add a column to the table for comments attached to each layer.

2.2 Layer Specification: Section 3.2

Within this section of the VC documentation, each of the system-level abstraction layers defined in the *Layering Principles* is to be described. The sections contained within Section 3.2 of the VC documentation include specifications for the:

- Data types used
- VC internal behavioral description (if one exists at this layer)
- General interface properties for this layer
- Specific VC interface descriptions with links to other interface-abstraction layers
- Links to the VC internal behavioral description (if appropriate)

These are detailed for the VC documentation in Section 2.3 through Section 2.8 of this document.

2.3 Data Types: Section 3.2.N.1

2.3.1 General Discussion

Data Types: Viable data types are specified by the *VSIA SLD Data Type Specification* (available in 1Q2000). Any types more complex than these must be fully structurally specified in documentation given by the VC Provider.

Data Formats: A data format is the structure of all information accompanying a datum that crosses a VC interface.

A datum is an object that assumes one of the documented data types. These data are passed through interface ports and may be transported and acted upon by messages and transactions. A message or transaction may affect the data format fields associated with the shipping of a datum, but may not affect the content of the datum itself (i.e., dataValue).

The data format that is be associated with a datum may include fields for the following information:

- dataValue – the value of the datum transferred.
- dataSize – the size of storage required for the dataValue field. This may be variable or a range at higher levels of abstraction.
- dataTag – for identification, an abstraction of addressing.
- dataPriority – the processing-priority related to the particular data object.
- dataTimeStamp – for communication of time between models. May be represented as a triple: (#_of_events, type_of_event, error).

These fields are not compulsory, and different combinations may be used at different levels of abstraction. For example, at a very high level of abstraction the dataValue and dataTimeStamp may be omitted. At the lowest level of abstraction in this document, Level 0.0, only the dataValue field is available.

Each of the fields described above must be described in a specific format. This format must be chosen and specified by the VC provider. The following are recommended: *dataValue* – dataType, *dataSize* – Integer, *dataTag* – Integer, *dataPriority* – Integer, and *dataTimeStamp* – (Integer, String, Double), where Integer and Double can only assume positive values in this context.

The dataValue field specifies the value and the type of the data to be transported over the port or by the transaction/message. If the dataSize required to support this value is variable, this must be defined in the section which describes the dataType adopted by dataValue. The documentation must explicitly reveal how the dataSize variability is exposed by the dataType. Note that the data format may be as simple as a dataValue field with only the data-size specified. This is the case when no value is associated with the object but there is an associated size required to model the data processing. This is common to performance models.

The dataTag field can be used to associate more than one scalar dataValue or subport to an aggregate dataValue, port or internal variable (see Section 2.8 regarding association to variables). For example, an array or memory of values can be associated with an indexed port and the placement of such a memory can be internal or external to the VC. The data type of the dataTag field will usually be of integer type, but can also be an enumeration of symbolic values. Typically, ports with dataTags are refined to individual ports (e.g., address ports) at a lower interface Layer 0.x.

The dataPriority and dataTimeStamp fields are associated with the VC's dynamic semantics of timing and concurrency (see Section 2.7.4). They are to be used to link the semantic models of the VC and the system. The data type of the dataPriority field will be an ordered type. Priority assignments can be hierarchically specified. That is, a port may be assigned a priority of access (see Section 2.7.1). When that port is accessed based upon its priority and behavioral context (e.g., the port is "triggered" or a "queue is non-empty"), the transaction/message through it may also be selected based upon a priority. Furthermore, the data accessed may also be determined on the basis of priority associated with the datum itself.

Dynamic priority assignment is supported through the association of priority to datum. A datum may have an assigned priority even if the associated port and transaction/messages have no statically assigned priorities.

A Layer 1.0 and Layer 0.x interface transaction/port may have all of the above fields associated with it. A Layer 0.0 message, however, only supports a *dataValue* field. Typing for that field is directly associated with a port. In general, the type and size of the data passed through an untyped port (above layer 0.0) can be different for each transaction/message. A transaction through a typed port cannot support a data type incompatible with that port. A datum cannot possess a type incompatible with a typed transaction/message which handles it.

A data format and associated data types described in this section may be referenced from:

- A *Port* - Every data transaction through this port passes information of this format.
- A *Transaction/Message* - Every transaction/message of this type through a port passes information of this format.
- A *Datum* - This is to allow the case of deferred-typing/formatting associated with a port or transaction. Under this circumstance, the elements of the protocol block cannot operate upon information contained within the format. If passed across a Layer 1.0 interface into a VC behavioral model, the datum will become associated with an IO variable within the behavioral model. It is the responsibility of the VC provider to have ensured deferred type/format handling within the internal VC behavior. The dynamic data-format associated with a datum can only be passed through a non-typed port, and can only be handled by a non-typed transaction/message.

Data Type Coercion or Translation: Data type coercion or translation, that may occur as a datum, is passed through interface layers. For example, a more abstract view of the type is broken down into more detailed structure. Typical coercions/translations may be described in this section. Others (specific to a certain interface and set of ports) may be given with the inter-layer static mappings (see Section 2.6.3.3).

2.3.2 Documentation Requirements

Note: It is mandatory that all types required for Layer 1.0 be defined in the section describing that layer. All subsequent definitions of data type and data format may reference this material. It is only mandatory to document all the new types and formats generated for each level of VC abstraction. All other types may be referenced. The names generated for each dataType should include an identifier for the layer at which it is defined. The recommended name structure is $dataTypeName_$Layer. A name not followed by a _$Layer post-modifier is assumed to be described in the Layer 1.0 data types section.

It is recommended that the data formats and data types used be partitioned into two parts as defined by usage, and described in sections:

- 3.2.N.1.1 General Types
- 3.2.N.1.2 Transport Types (used at the interface)

The description consists of:

- Names of each dataType used to describe a data value or field of a dataFormat.
- Description (or reference) for the type.
- Name of each dataFormat used.
- Structure of each dataFormat and declaration of the dataTypes associated with each field.
- (optional) Set of typical type coersions/translations.

2.4 *Internal VC Behavioral Description: Section 3.2.N.2*

2.4.1 General Discussion

The only layer for which this description is mandatory for is Layer 1.0. For all other layers, the provision of a VC behavioral model is a choice left to the VC provider. Even at Layer 1.0 the VC provider is at liberty to only provide a block-box description of the component. The description of the behavior must reveal sufficient information about the externally visible operational properties of the VC to permit connectivity into a system.

The minimum level of specification required of the VC behavior is functional identification (description of the purpose of the block), and identification of the IO required to perform that function. As such, each of the input/output variables associated with the operation of the VC must be named and categorized. Categorizing requires the specification of all data format and typing requirements and identification of the purpose of each input and output, if assignment and interpretability of data is required of the environment.

This specification must further be extended to basic actions passing across the VC interface which may be associated with control-type actions (e.g., trigger or emit).

Note: The relationship between variable sequences and actions cannot imply constraints which the protocol block must interpret for valid operation. Any operation/variable exposed as an IO must be able to be interpreted individually by the environment (a requirement for behavior/interface dissociation).

Recommendations for the detailed specification of the internal behavior of system level VCs will be defined by the VSIA SLD DWG. Now we are only concerned with the internal behavior insofar as it affects the external view of the VC.

2.4.2 Documentation Requirements

For each IO variable required of the VC behavior:

- Provide a name for the variable or action.
- Provide the data format or data type associated with any variable.
- Categorize the variable (sufficient description to allow adequate interpretation).

For each IO action required of the VC behavior:

- Name the action.
- In the case of an action (e.g., trigger), specify the form of sensitivity required.
- Categorize the variable (sufficient description to allow adequate interpretation).

2.5 *Interface Overview and General Properties: Section 3.2.N.3.1*

2.5.1 General Discussion

Section 3.2.N.3.1 is one of two primary components of interface description Section 3.2.N.3 of the system-level VC documentation. The other component, Section 3.2.N.3.2, is discussed in Section 2.6.

Before describing the specific port behaviors for each interface at an abstraction layer, a set of general assumptions may be specified. These assumptions are a set of interface behaviors assumed of all ports/ transactions of a particular type. In particular, this specification of general properties must be given up-front when the interface model is constrained within the operational bounds of a model of computation (e.g., data flow). The set of implied control attributes associated with each transaction type must be described here.

These operational assumptions may be any of a set of behaviors which may be described as associated with a single port at this abstraction layer (e.g., blocking attribute and so on). However, the set of multi-port common behaviors which may be specified here are restricted to:

- *TriggerAny* – Generate a VC trigger when any input arrives at the VC.
- *TriggerAll* – Generate a VC trigger when all inputs arrive at the VC.

In this section, the VC provider must also provide a description of how many interfaces the VC has at this abstraction layer, and how/why they are specified separately. To define separate interfaces, the VC provider should cluster ports according to common themes. For example, at Layer 1.0 one may choose to cluster the input data ports into one interface, the output data ports into another interface, and all the control/trigger ports for the behavior into a third interface. At more refined layers of abstraction, the port clustering into interfaces would follow common protocol clustering practices; for example, interrupt interface, bus interface, DMA interface, etc. In general, this partitioning will help the VC integrator gain a better understanding of the communication strategy used by the VC.

2.5.2 Documentation Requirements

- Depict the VC with all interfaces. The sketch of the VC should include all the interface names, and may show the ports and port-names if the interfaces are sufficiently simple.
- List the names of all interfaces defined at this layer of abstraction.
- For each interface, describe (or reference) a purpose for its existence.
- Using the attribute (Section 2.7.1) and transaction (Section 2.7.2) types, describe any operational properties common to all ports at this layer of abstraction (e.g., all read ports are blocking, all write ports are non-blocking).
- Using the attribute (Section 2.7.1) and transaction (Section 2.7.2) types, describe the set of channel properties assumed to be supported by the environment (e.g., all channels support infinite FIFO buffering, and so on).
- If the interfaces at this layer are not directly associated by name to the interfaces at the more abstract interface-layers, show the correct association in tabular form.

2.5.3 Documentation Options

The port attribute description technique for interfaces is given in Section 2.7.1. This provides a visual shorthand for representation of port intent (e.g., data / control, blocking/nonblocking, etc). For the abstract interface layers with few ports per interface, the VC provider may choose to represent all ports and port-attributes in this part of the document, Overview and General Properties: Section 3.2.N.3.1, rather than in: Behavioral – Port Attributes: Section 3.2.N.3.2 B1 $Interface Name.

2.6 Interface Structure

The descriptions of interface structure fall into the following categories:

Single-Port Descriptions
In this section of the document, the ports are defined and typed (see Section 2.6.2).

Multi-Port Descriptions

In this section of the document, the static association of ports from a more abstract level to the more refined is specified. There are two aspects that drive this static-association – further refinement of behavior, or further refinement of data types (see Section 2.6.3). Although the structural association is defined here, the form of behavioral refinement is not. This is a dynamic property and as such is described later in a section following a behavioral-typing of the ports.

2.6.1 Interface-Specific Description: Section 3.2.N.3.2 $InterfaceName

Prior to the detailed description of structure and behavior, it is recommended to the VC providers that they provide a short verbal description of the structure of this interface and its operational principles. No particular format is given for this material.

2.6.2 Structural – Port Identification: Section 3.2.N.3.2 S1 $InterfaceName

2.6.2.1 General Discussion

A port is an externally visible connection point of the VC interface. All channels are connected to a VC interface through these ports. This section lists the names and gives the static definitions of the ports in the VC interface at this level of abstraction. Apart from the name of the port, the type of port is defined as well as the role of the port.

Ports are not strictly defined in the roles of data and control but may be more generally described as a *producer* or *consumer* of data, or an *initiator* or *responder* for control information. *Production* and *consumption* require the existence of a defined state, whereas *initiation* or *response* does not refine the form of information sent or received to a specific form. Note that *initiator* and *responder* do not in themselves designate a master/slave relationship, as only the commencement of action is identified.

At lower levels, a port may be identified as *in*, *out* or *in_out*.

2.6.2.2 Documentation Requirements

There are two main parts to this section:

- Part 1: An alphabetic listing (in a table) of each port name in this layer, with its role, its data format (see Section 2.3), and with a cross-reference to a table row or paragraph number in Part 2. Optionally, references to other paragraphs containing additional information about this port can be included. The data format may be fully specified in the table, in a reference to a data format defined elsewhere, or immediately after the table. It may be convenient to define the type of the dataValue in the table and refer to a common data format defined in an earlier section for the other fields. Additionally, if an executable behavior exists and is provided at this level, then an additional column giving the name of the equivalent port on the behavior component must be provided.
- Part 2: A series of table rows or paragraphs headed by one or more port names and containing a short description briefly outlining the purpose of the ports. The names in Part 2 may be given in any order chosen by the writer.

Data and control ports that are explicitly associated may be identified by names with the same root. It is recommended that the full port name includes a reference to the interface and layer of abstraction in which it is defined as in $PortName_$InterfaceName_$Layer, so that any reference to a port in another part of the document is not ambiguous.

Table 4 and Table 5 give examples in tabular representation of the port data. This is broken down into two tables to represent a very general formatting style, but a single table will often suffice. The two-table method allows a single description to apply to multiple-ports.

In these tables, the *Role* of the port indicates its producer/consumer action in terms of data, or its Initiator/ Responder action in terms of control (refer to Section 2.7.2). The *Data Format* will generally make reference to a format described in the Data Types: Section 3.2.N.1 of the VC documentation. The *Description* is intended as a short introduction to the purpose and operation of the port, and should list key properties, such at port priority, etc. The *Behavior* is a cross reference to the detailed description of the port behavior which would normally be given in Behavioral – Port Transactions: Section 3.2.N.3.2 B2 $InterfaceName. This pointer is only a documentation requirement if the behavior is to be cross-referenced to a location other than the expected section (i.e., to an earlier layer description that passes down directly to this layer).

In this example, an executable does not exist at this level of abstraction.

Table 4: Structure of the Port Definition

Port Name	Role	Data Format	Description	Behavior
A_port_0_7	Responder	TimeStamp => TsType_0_3	Row 2	Section 3.2.1.3.2, on page L
B_port_0_7	Producer, Initiator	See Data Format 1 below	Row 1	page M
		Value => BIT		
C_port_0_7	Consumer	Others => see Df_C_0_3	Row 2	page N

Data Format 1:

Value => type_A_0_3

Tag => type_gTag

Priority => type_gPriority

Table 5: Structure of the Port Description

Row	Port(s)	Description
1	B_port_0_7	This port is used by the component to ask questions ...
2	A_port_0_7 C_port_0_7	These ports allow potential answers to be checked ...

If a port has no type, then the transactions that pass through that port will give the type of the data through the port.

2.6.3 Structural – Inter-Layer Static Mappings: Section 3.2.N.3.2 S2 $Interface-Name

In Section 3.2.N.3.2, the static mappings between the ports at the next higher level of abstraction and the ports at this level are described. This includes the mappings between the types of the ports at the different levels. The dynamic or behavioral mappings are described in this document in Section 2.7.3, Inter-Layer Behavioral Mapping. Of course, these inter-layer sections do not make sense at Layer 1.0 where they can be omitted or, optionally, a description of important temporal and/or functional requirements and their mappings to the VC interface can be inserted, as described in Section 2.7.3.

2.6.3.1 General Comments – Port-to-Port Mapping

The static port-to-port mappings simply document the existence of a relationship between a port at the higher level of interface abstraction and ports at this level. This is used to structurally link the interface abstraction layers. However, this does not describe how the set of refined ports implement the behavior associated with the more abstract ports. This is the role of a later section in the interface documentation associated with interface behavior, described in Section 2.7.3. For each structural linking of an abstract port to a set of more refined ports, there must be a corresponding mapping in the inter-layer behavioral mapping which describes the behavioral relationship between these ports. Note that more than one abstract port may map to a lower level port. The sharing of this port is defined in the behaviors and protocols for the ports.

2.6.3.2 Documentation Requirements – Port-to-Port Mapping

The documentation of these relationships is a simple one-to-many mapping in a table format, with the more abstract port at the head and the ports at this level of abstraction following. It is recommended that a cross-reference to the section that describes the dynamic mapping also be given. An optional short summary description of the relationship may also be captured. Note that this strategy extends to many-to-one mappings. For example, a many-to-one mapping from {A,B,C} to X would have three rows mapping A, B and C individually to X. Table 6 provides an example of this tabular-representation.

Table 6: Structure of the Port Static Mapping

Abstract Port	Refined Ports (at this level)	Behavior Mapping	Comment
A_port_0_7	CS, CLK	<section x.y.x> on page <p>	Implement Port_A's protocol fully
B_port_0_7	RD_REQ, RD_ACK, ADDRESS_port_0_2	<section x.y.y> on page <p>	Port_B refined to 4-16 bytes/clk
C_port_0_7	CLK, DATA_port_0_2	<section x.y.z> on page <q>	Port_C's shared protocol

Figure 8 illustrates mapping between two layers.

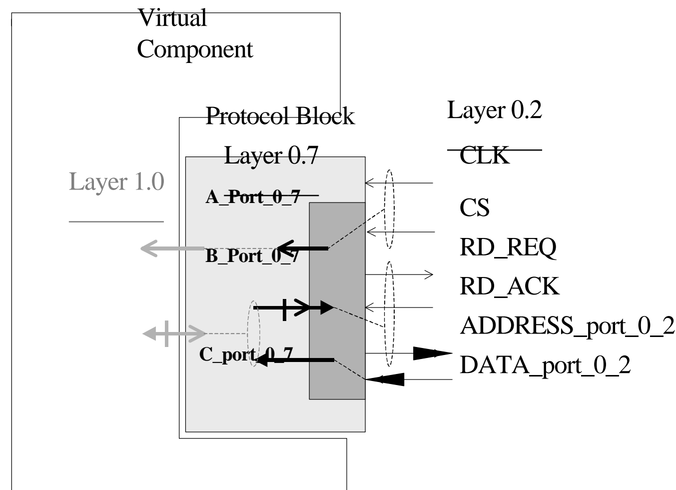


Figure 8: Example: Layer 0.7/Layer 0.2 Ports for a VC

2.6.3.3 General Comments – Data-Type Mapping

Apart from ports, the other static change in interfaces between levels of abstraction is the gradual refinement of data from a more abstract type to a more implementable (physically or machine code) data type. This section defines the relationship between the data types and formats (see Section 2.3) of the ports and transactions at the more abstract level and the level through which the actual data may be transformed (if such a relationship exists separately from the behavioral mappings defined in Section 2.7.3). Note that a many-to-1 abstract-to-refined mapping cannot have only a data mapping. This abstract must have a behavioral mapping.

There are two kinds of data-type mappings:

- Type coercion – required when the data are in compatible formats but of different types.
- Type translation – required when the data-values are incompatible.

Note: These refinement mappings are not permitted between the internal VC behavioral model and the Layer 1.0 abstraction level. At this level, VC functionality must handle unique identification of each object brought through any Layer 1.0 port. For example, if the VC operates on complex numbers with the *Re* part brought through port X and the *Im* part brought through port Y, the functionality of the VC must include the reassembly of the separately-identified variables. For example, *complex_number(dataTag) = {number(dataTag).Port_X, number(dataTag).Port_Y}*

Type Coercion:

When abstract ports are mapped one-to-one or one-to-many to refined ports, their types may not be exactly the same. However, the mapping is valid if the types are compatible as defined by the rules of the VSIA SLD data types documentation. If the types are compatible, then no extra information need be given here. The ports can simply be mapped (as in the Section 2.7.3 Behavioral Mapping). Note that type compatibility is commutative and associative, that is, if A is compatible with B, then B is compatible with A. Further, if B is also compatible with C, then A is compatible with C. These relationships are important properties for level changing in the interface.

A discussion of some of the relevant parts of the SLD DWG data type specification follows. The data types associated with the ports in a port mapping have to be compatible. With respect to the types specified by the SLD DWG data type specification, type-hierarchy (informal) and compatibility are defined. These include conversions between numerical types and (uninterpreted) bit vector types. For these types, the compatibility and the mapping between the data types and the corresponding ports may also be defined using the operations and other manipulation functions specified by the SLD DWG data type specification. Examples are the subvector and concatenation operators, which can be used to split a type of size n into individual elements of size m.

The following are examples of type-hierarchies:

- vsi_int32 is compatible with integer
- vsi_signed(24) is compatible with integer
- vsi_bitvector(24) is compatible with vsi_signed(24)
- vsi_bitvector(24) is incompatible with vsi_unsigned(12)
- vsi_bit is compatible with boolean

These examples are to be extended in the natural way towards the unconstrained types of integer, double, etc.

The same guidelines apply for the use of a record at a higher layer where a dataTag field exists to address the individual fields and the refinement to individual ports for each field. Vice versa, the mapping can be used for time multiplexing more than a dataValue on a single port. For example, the first transaction involves transporting the Re part of a complex number, and the second transaction involves the Im part of that number. Such a port refinement has to be accompanied by the behavioral associations of these layer, sequencing, and concurrency requirements and assumptions (see Sections 2.7.3 and 2.7.4).

Data-type compatibility is restricted to the ports that make up the dataValue component of a transaction at a higher Layer 0.y or Layer 1.0. The control ports (e.g., the r/w indication when multiplexing a data-in and data-out port) introduced or modified in this refinement are not considered.

Examples**:**

- Two ports at Layer 0.x of type vsi_bitvector(8) are compatible with a port at higher Layer 0.y of type vsi_int16.
- A port at Layer 0.x of type vsi_bitvector(32) is compatible with a port at a higher Layer 0.y of type struct {vsi_signed(16) Re; vsi_signed(16) Im}. The fields in the latter structs are “subports” at the Layer 0.y.
- With the associated sequencing and concurrency requirements, two ports at Layer 0.x of type vsi_bit and vsi_bitvector(8) (inout) respectively, are compatible with two unidirectional ports (in and out) at higher Layer 0.y of type vsi_int16.
- Two ports at Layer 0.x of type vsi_bitvector(32) and vsi_bitvector(8) respectively are compatible with a port at a higher Layer0.y for which the dataValue has type Integer, and which also has a dataTag of type vsi_int8.

Type Translation:

Sometimes, ports may have a one-to-one or one-to-many mapping, but the types may be totally incompatible without interpretation. The interpretation would allow translation between two otherwise incompatible types. This has to be provided by the interface writer. The translation can be done in two ways, by pattern matching or by subprogram. Note that the translation has to be provided in both directions, to maintain the properties of commutation and association required for arbitrary level changing. However, due to the greater amount of information captured at the more refined levels, it is often difficult to preserve information when translating types. Therefore, information preservation is not a requirement. If information is not preserved, a suitable comment or annotation is recommended.

Pattern matched interpretation would allow the translation of one type to the other and vice versa, by providing matched lists of patterns in both types. This form of translation, by its nature, is declarative and is thus reversible. An example of such an interpretation is a table with an enumerated type in one column and a bit-pattern in the other column. Each possible enumeration and all possible patterns of the two types should be covered. A default mechanism is allowed.

Subprogram translation needs two subprograms – one to translate from the more abstract type to the more refined type, and one to do the reverse translation. The name of the subprograms should clearly indicate their intent, and their content should be documented in this section in high level pseudo-code or literate programming.

2.6.3.4 Documentation Requirements – Data-Type Mapping

For port-port type mappings using type coercion, the following information should be provided: a list containing the abstract port aggregate and its type, the refined port aggregate and its type, and references to the appropriate sections of the VSIA SLD Data-Types. For mappings using data translation, there may be more than one abstract and refined port aggregate. In this case, an additional reference is needed to the data translation mechanism and to where it is defined in the Data-Types section.

Naturally, for each mapping, the ports in the aggregates should also be in the appropriate row in the port static mapping table, as shown in Table 7.

Table 7: Data-Type Mapping Using Data Translation

Abstract Aggregate	Type	Refined Aggregate	Type	Translator	References
Port_X_0_2	Vsi_signed(8)	Port_A[0 to 3] & Port_B[4 to 7]	vsi_bitvector(8)	None – coercion	VSI SLD DTD
Port_Y_0_2, Port_Z_0_2	Boolean, Integer	Port_C[0 to 31], Port_D	vsi_bitvector(32), vsi_bit	Cond_Mask_translate	this doc 3.2.2.1:4
Port_E	A_type (enum)	Port_G & Port_H	vsi_bitvector(2)	Enum_Mask_translate	this doc 3.2.0.1:6

As suggested in the General Comments paragraph, the data translation mechanism would be documented by giving the name of the translator followed either by a table with type pattern matches or by the names of two subprograms that perform the type translation. The bodies of the subprograms can either be documented inline or in an appendix. A comment should be written if the translation is unclear in either direction. For example:

Enum_Mask_translate:

A_type	**vsi_bitvector(2)**
Fred	00
Eric	01
Joe	10
John	11

Cond_Mask_translate:

Down Cond_Mask_translate_down(boolean, integer) => vsi_bitvector(32), vsi_bit

Up Cond_Mask_translate_up(vsi_bitvector(32), vsi_bit) => boolean, integer

2.7 Interface Behavioral Descriptions

Every VC has a set of requirements that must be met and a set of properties that can be utilized for the integration of that component into a system environment. These requirements and properties are the specifications that ensure that the use of the component satisfies the expected abstract behavior. In the *Behavioral Documentation Standard*, input and output behaviors of a VC are described through a set of transactions, messages, and attributes [4]. The attributes, when applied to the inputs of a VC, imply a set of requirements upon the communication channel and interconnected VCs. Applied to the outputs, these attributes specify the richness of the communication protocol that the VC behavior can support. For example, if a VC, which does not support blocking-write, is connected to a VC which requires a blocking-read behavior, then the channel connecting the two VCs must provide suitable services to enable this communication (e.g., Queuing).

The basic principles involved in describing the behavior of a layered interface is as follows:

All activity through a port of a Layer 1.0 Behavioral Interface is referred to as a transaction. The activity over the component interface (*Layer 0.0*) is composed of messages. Transactions are not required to be atomic. Messages are required to be atomic. Atomic is defined as follows:

> An **atomic action** either completes fully or not at all. An atomic action which completes fully does so without the possibility of interruption or interference. If an atomic action does not complete fully (i.e., is abandoned) then the state of the system must be as if the action had never started.

The transaction set for Layer 1.0 activity is very small. This is enforced to ensure that at least one sufficient abstract layer of an interface description is provided with every VC. The transaction set allowed for all 0.x layers of interface abstraction above Layer 0.0 is somewhat broader than that permissible for Layer 1.0 descriptions.

Every transaction or message can have an *attribute*, or set of attributes, which help describe its basic behavior. These attributes are associated with each of the ports through which the transactions operate. For example, a particular transaction may or may not be interruptable, may or may not be an externally triggered operation, or may or may not require an input value at a port to be persistent, etc. The set of attributes that can apply to messages is very simple, as these actions must be atomic.

Commencement of a sequence of messages associated with a particular transaction does not necessarily prevent the commencement of further transactions. An example is depicted in Figure 9. The ability to overlap message sequences is implemented by the protocol block.

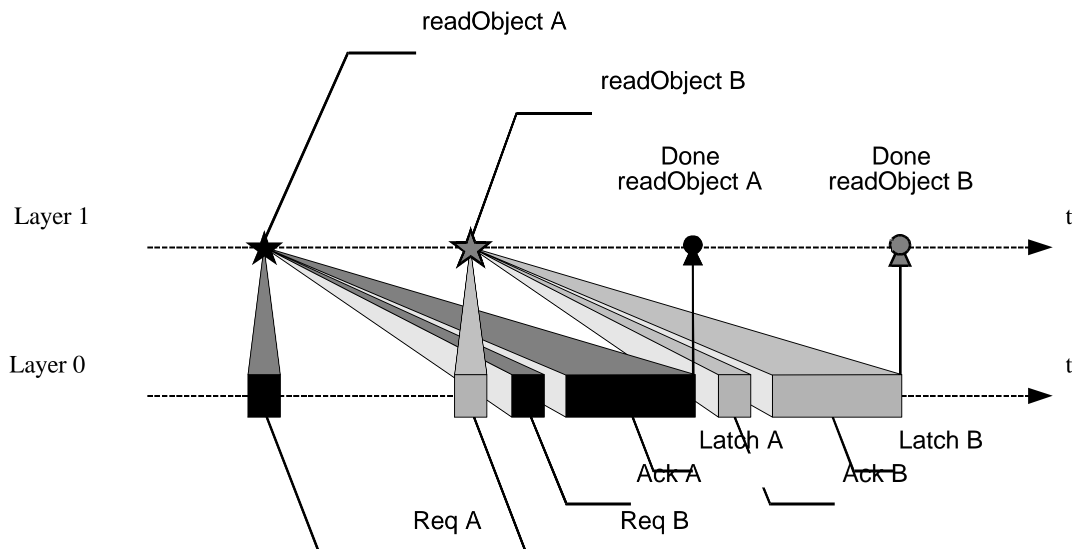
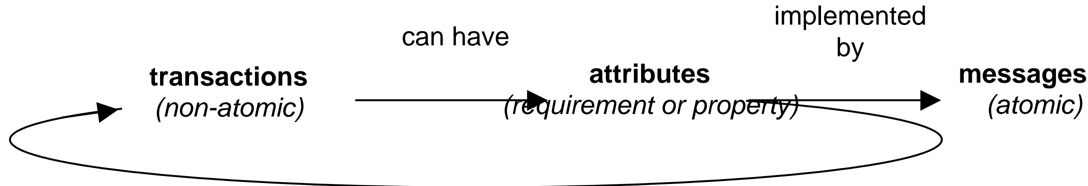


Figure 9: Interleaving of Message Resulting from Sequential Transactions

In summary, the relationship between transactions/attributes/messages across the interface abstraction layers is:



This section of the interface documentation is broken into single and multi-port views of the interface description. documentation of the interface proceeds as follows:

Single-Port Descriptions

The behavior in terms of attributes supported by each port is covered in Section 2.7.1 Behavioral – Port Attributes: Section 3.2.N.3.2 B1 $InterfaceName. The transactions permissible through each port is covered in Behavioral –Port Transactions: Section 3.2.N.3.2 B2 $InterfaceName (See Section 2.7.2). No multi-port behaviors are specified here. By specifying the types of behaviors compatible with the ports, this is effectively a behavioral typing by which the issue of compatibility during connection may be assessed.

Multi-Port Descriptions

Multi-ports are covered in two sections, Section 2.7.3 Behavioral – Inter-Layer Behavioral Mapping: Section 3.2.N.3.2 B3 $InterfaceName and Section 2.7.4 Behavioral – Protocol Description: Section 3.2.N.3.2 B4 $InterfaceName. These sections must utilize the organization hierarchy specified in terms of structural association of ports (see Section 2.6.3 Structural – Inter-Layer Static Mappings: Section 3.2.N.3.2 S2 $InterfaceName). In *Inter-Layer Behavioral Mapping* the inter-relationship between behaviors specified at the various interface abstraction-layers is shown. This section must detail the requirements (which include the set of assumptions) and properties of the more abstract layers which are passed down as constraints to the current layer. In the second related section, *Protocol Description*, all inter-relationships between port behaviors must be described. In part, this is to show that the properties required of the *Inter-Layer Behavioral Mappings* are satisfied. Included also are any other inter-relationship of ports specific to this abstraction layer. From this description are derived a further set of assumptions, requirements, and properties which are passed down to the next most detailed layer of interface abstraction.

2.7.1 Behavioral – Port Attributes: Section 3.2.N.3.2 B1 $InterfaceName

2.7.1.1 General Comments

When information is transferred between two virtual components, an information flow is created and a set of requirements for a physical channel is defined. Information could imply either the setting and receiving of a specific state or the causing and detection of change-of-state. We distinguish data flow to be that flow requiring state information, and control to be that driven by sensing of change-of-state. At the highest level of abstraction, Layer 1.0, the physical channels do not actually exist, and all information flow is defined as point-to-point communication over a perfect channel. At the highest layer, we must address issues such as the attributes of the information flow, the control semantics associated with it, the persistence of data, and many other functional issues. As the design is refined, timing issues become more important and physical constraints on some of the parameters need to be imposed. Eventually, all of these properties will be reflected in the physical design.

In Behavioral – Port Attributes: Section 3.2.N.3.2 B1 $InterfaceName, the attributes of the information flows are defined along with a graphical notation to allow the important information about the external interface of a VC to be presented in an intuitive manner. Information flow is the data and control transfer between any two functional blocks (see Figure10). This graphical notation is not intended for use in a system-level modeling tool or simulator. It is intended to give a quick and reasonably clear indication of the primary interfaces surrounding a virtual component. However, the diagram should not be considered as a complete definition of the interface, and all interfaces should be backed up by a complete textual description. These documentation requirements are expanded in more detail in Section 2.7.1.6.

The attribute set as defined can be used either to depict the functional aspects of an interface or to define the capabilities or services that can be provided by a pure architectural model, such as a microprocessor model or a memory unit. It should be possible in the design process to match the services provided by these architectural models with the communication attributes defined at the functional level. Any mismatches found may result in either a change to the functional attributes required or additional services being provided by an architectural block. During the design process all of the high level communication attributes need to be satisfied by services, either in the implementation of the ports or channels or incorporated into the functionality of the blocks on either end of the communication link.

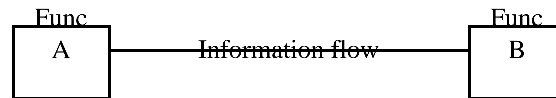


Figure 10: Information Flow

Attributes can be divided into one of three primary types. These are data attributes, control attributes, and sequence or timing attributes. At Layer 1.0, no timing information is presented but sequencing information may be required for correct operation. As the design is refined and the functionality is mapped onto physical channels, timing becomes more important. However, there is a middle ground where we may wish to add approximate or estimated times to information flows before full architectural modeling has been completed.

For some attributes, default actions have been assigned for end cases. If these actions do not fit with the required behavior, they can be overridden by fully describing the actions for the flows in the supporting documentation.

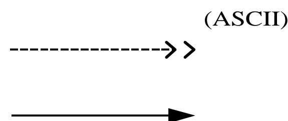
An attribute is associated with a port. Any transaction or message through a port supporting a specific attribute will inherit the behaviors of that attribute. Therefore, it is not permitted to have a single port that supports different attributes dependent upon the transaction or message executed through it.

2.7.1.2 Layer 1.0 Interface Abstraction

Data Flow

Basic Data flow

A flow of data from one block to another is depicted by a solid arrowhead, pointing in the direction of the flow of data.



There is no typing of data on the interface itself. Typing information is derived from the port or protocol, which is specified separately. Additionally, the interface does not restrict the flow of a single type of data. Neither does it require that the actual data be present. For example, it is acceptable to have the interface define how much data of a particular type will flow, but for there to be no explicit data contents.

There are no control semantics associated with a basic data flow. This means that the receiving block will receive no notification that data has been placed on the flow. Additionally, no input sensitivity is assumed for the receiving block.

At Layer 1.0 all communication is point to point and bi-directional data flow is not supported. The only transactions supported at Layer 1.0 for a pure data flow are *transReadObject* and *transWriteObject.*

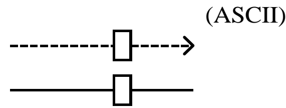
Data Persistence

When a piece of functionality makes data available on a basic data flow, there is no persistence associated with that information. This means that the data is only valid and available at the instance in which it is provided. Thus, if a receiving block were to attempt to read a basic flow's contents at some other time, no data would be present. There are two methods to ensure that the data that is read is valid and available. The first is to combine the data flow with a control flow. This is described later. The second method is

to add persistence to the flow. This can be done with one of two constructs. The first is to buffer the data and the second is to add a FIFO. Both of these are described more fully in the following sections.

Buffer

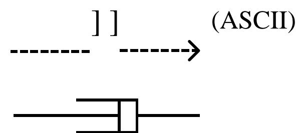
A buffer can be added to a basic data flow to add persistence. Adding an open box on a data flow shows the inclusion of a buffer.



When data is read from the buffer, the contents of the data remain unchanged. Thus, subsequent reads with no intervening writes will continue to return the same data. Data written to the buffer overwrites the existing data, which may result in data loss if the data is not read before the second write. While the buffer is shown to be on the interface, this does not imply anything about the eventual location of the implementation of the buffer. As part of the design process, the necessary functionality may be incorporated into the sender of the data, the receiver, or in the channel itself.

FIFO

A second method to implement asynchronous communication with persistence is by using a FIFO. This ensures that no data is lost across the interface, but there is clearly going to be a higher implementation cost. The presence of a FIFO is indicated by the following symbol added onto a basic data flow.

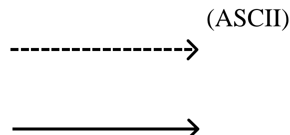


At Layer 1.0, FIFOs are of infinite depth. Thus, any write to the device will result in the next space in the FIFO receiving the data. When reading data, all data is read in the order it was written. If empty, a read will result in no data available in the same way as for the basic data flow.

Control Flow

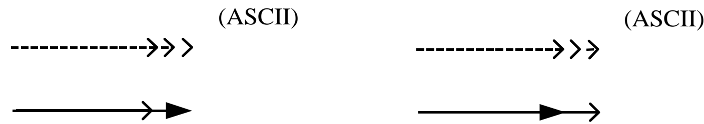
Basic Control Flow

A flow of control causes an action to occur in another behavioral block. Unlike the data flows described above, there is no data associated with a control flow, although both can be combined into a single set of attributes on a flow. A flow of control is depicted by an open-ended arrow.

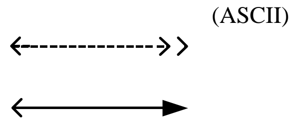


The arrow points toward the behavioral block where an action is to be initiated and away from the behavioral block that caused the initiation. Input sensitivity is assumed for all flows with control attributes. The basic control flow is nonblocking, which means that as soon as the message has been passed, the initiator will resume execution and will not be blocked by the execution of the receiver. Thus both behavioral blocks will proceed in parallel. A pure control flow at Layer 1.0 may have *messSense* and *transEmit* transactions associated with it, but may not have data associated with the transactions. This means the only valid event type is an *anyEvent*.

Combining a control flow with a data flow enables a synchronous flow of communication. Two basic types are possible. The first type, where the control flow and data flow are both in the same direction, indicates that whenever data becomes available on the interface, a control message is given at the same time indicating that the data should be read. There is no difference based on the ordering of the arrows, as depicted below.



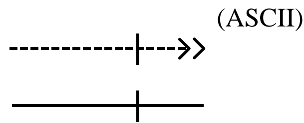
The second type of control flow/data flow combination is where the flow of control and data are in different directions.



This indicates that the data flow was caused by the action of the receiving behavioral block. Thus, while the left-hand side is the data master, the right-hand side is the control master. However, the transaction is still nonblocking, which implies that there is no guarantee of data validity or availability when it is requested. All Layer 1.0 transactions are available for these mixed control/data flow channels, as well as all defined event types that are supported by the data types on the ports connected to the channel.

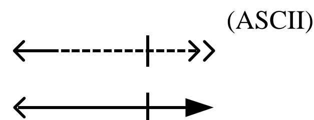
Blocking

As indicated above, a basic control flow is nonblocking in nature. If a blocking flow is required, then adding the following symbol onto the interface will indicate this.



When a blocking attribute is added, the operation does not return control to the master until the request has been satisfied.

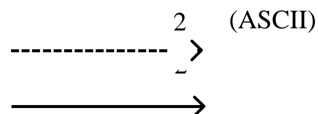
Consider the following set of attributes.



This indicates that a control operation is initiated from the right hand side with the intent of receiving some data from the left-hand side. The operation of the right-hand object will be blocked until the data is available and the left-hand side releases control back to the right hand side. Thus we are guaranteed to get valid data returned.

Control Priority

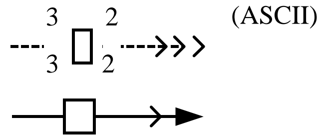
When a functional block receives more than one control flow, it is sometimes necessary to be able to define the order in which those control flows should be executed if they arrive simultaneously. Adding a number representing a relative priority close to the receiving end of a control flow indicates the priority of a flow. For example:



This would indicate that any flow with a priority of 1 would be executed first and then this control flow would be executed. Priorities can only be associated with a flow that has control attributes defined. It is important to separate this notion from data priority, which is described later.

Multi-rate Systems

One special case that needs to be considered is a multi-rate system. In these systems, which are essentially data flow systems with implied control semantics, each functional block fires when there is adequate data on all of its inputs. Since many of the blocks require different amounts of data before execution can begin, each of the blocks will be running with a different period. For Layer 1.0 interfaces, these data requirements are specified with an extension to the data buffering scheme described above under the data flow attributes. The buffer is assumed to be of infinite depth.



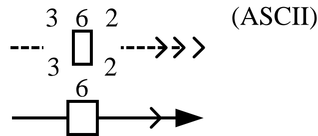
In this example, the sending side will produce three pieces of data each time it executes and the receiving side will extract two pieces of data each time it executes. The control flow will fire whenever there is sufficient data in the buffer to satisfy the receiving side.

2.7.1.3 Layer 0.x Interface Abstraction

As the design is being refined, additional attributes may need to be defined on interfaces. These attributes deal with timing and with sequencing, as well as with other more complex constructs necessary to define the details with which the information exchange is to take place. All of these constructs build on those defined for Layer 1.0.

Buffer with defined depth

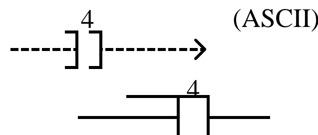
The multi-rate buffer at Layer 1.0 was assumed to be of infinite size. Clearly this is not practical. The size of the buffer must be constrained as the design is refined. Placing a number within the buffer symbol indicates the size of the buffer.



In this particular example, two sets of data could be placed into the buffer before data is lost. This would provide for three sets of data available at the receiving side.

FIFO with defined depth

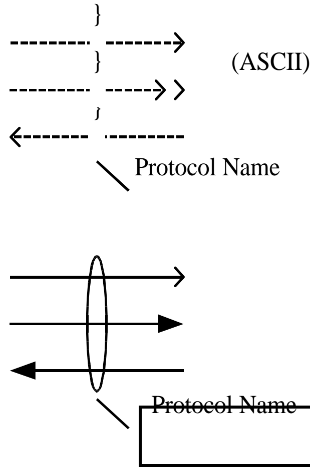
Similar to the buffer, the size of the FIFO defines the maximum amount of data that can be held before data is lost or the device blocks. Placing the depth number within the FIFO symbol indicates the depth of the FIFO.



A FIFO is non-blocking by default, so when a data overflow condition occurs, the last element of the FIFO is overwritten. If the desired action is for the write to block, then the blocking attribute can also be added to the flow.

Protocols

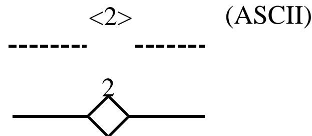
There are many cases, particularly as the design and the interfaces are refined, when it becomes impossible to define all of the attributes of a flow on a single communication path or channel. In these cases multiple flows will be defined with a protocol, which binds the behaviors of the flows. The existence of a protocol, and the flows governed by that protocol, is shown in the following diagram.



The protocol description defines the relationship among the flows, as well as which exchanges may or may not be legal. This association shows the grouping of transactions and ports that execute a complex protocol. The purpose and operation of this grouping must be further detailed in the section of the system-level VC documentation that covers the interface behavioral protocol description (see Section 2.7.4).

Pipeline

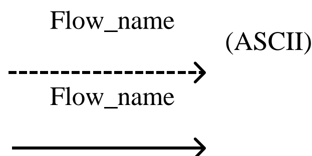
Another very common implementation attribute is a pipeline. Pipelines are much like FIFO buffers, except they have very specific and implementation dependent control semantics attached to them. Because of the importance of being able to quickly identify the existence of a pipeline, the special symbol below can be placed on a flow.



The number within the symbol indicates the number of pipeline stages present on the flow. It is highly recommended that all pipeline flows include a protocol, as described above, to indicate the control signals associated with the pipeline. The protocol would also be used to cross-reference the timing and sequencing information (given with the protocol description) that is present in the pipeline. (See Section 2.7.4.)

Delays

In order to specify delay information, the flow should be named. There are two ways to name a flow. The first as indicated for protocols above, except it includes only a single flow. The second is to attach the flow name directly onto the flow symbol as shown below.



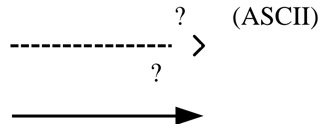
In the supporting documentation for the attributes, the flow_name should be linked to the correct portion of the documentation as described in Section 2.7.4 Behavioral – Protocol Description: Section 3.2.N.3.2 B4 $InterfaceName.

Exceptions

Exceptions are a special case of a protocol and should be described in the same way.

Data Priority

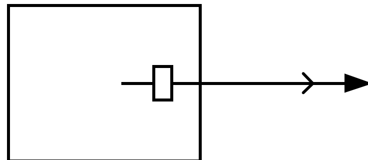
All of the previous transaction attributes have been static in nature; therefore, they are set on the channel. However, data priority is a dynamic attribute of the communications data and cannot be defined in a static manner. However, it is important to be able to identify those flows where data priority is to be applied. Such identification is accomplished by adding a question mark onto the data flow symbol as shown below.



Data priority should not be confused with control priority, described earlier, which is associated with the control aspects of the transaction. The details about the priority scheme being used should be documented along with the data type definitions for the port, as described in Section 2.6.2.

Functional Replacement of Buffers, FIFOs and Pipelines

As the design is refined, the implementation of buffers and FIFOs may be incorporated into the functionality of either the sender or receiver. However, since this design was originally specified as an interface functionality, it is important that this information not get lost. (Documentation should continue to include this information in lower level descriptions.) The diagram below shows how a buffer that is to be implemented as part of the sender should be diagrammed. It should be stressed that this migrated functionality still remains part of the system-level VCI and never becomes part of the actual functionality of the sender or receiver. Such an occurrence would contradict the attributes defined at the higher level of abstraction.



2.7.1.4 Layer 0.0 Interface Abstraction

Layer 0.0 corresponds to the implementation level for the documented IP and may exist at any level of abstraction. However, a fully described interface at the RTL level, which is likely to be the usual abstraction for Layer 0.0, should have all attributes fully resolved into functional blocks. These blocks should be described in either the sender or receiver VCI. In this case, the only flows left should be pure data flows, since the protocol section covering those flows will now describe all control semantics.

2.7.1.5 Behavioral Typing

When a port is connected to a net, the attributes must be compatible with all other ports connected to that net. Additionally, there are operational requirements for nets that must be matched by at least one of the ports connected to a net. For example, every net must have at least one data producer, one data consumer, and one port that will initiate a transaction. These requirements are shown in Table 8.

Table 8: Behavioral Typing by Attributes on Ports

Attribute on VC Port	Requirement on Connected Net	MUST NOT exist on Connected Net
Data – *Producer*	Data – *Consumer*	No Restriction
Data – *Consumer*	Data – *Producer*	No Restriction
Control – *Initiator*	Control – *Responder*	No Restriction
Control –*Responder*	Control – *Initiator*	No Restriction
Blocking – *Producer*	Non-Blocking – *Consumer*	Blocking – *Consumer*
Blocking – *Consumer*	Non-Blocking – *Producer*	Blocking – *Producer*
Data-Priority – *Producer*	Data-Priority – *Consumer*	No Restriction

For all other attributes listed, there are no necessary conditions implied on the connected net.

2.7.1.6 Documentation Requirements

Note: When a virtual component may present more than one attribute set for a particular interface, each of the possibilities should be described separately. For clarity, these attribute sets should be described in logical groups (such as, attribute set for boot sequence, attribute set for run-time communication, …). These groups may be named and switching between the attribute groups completed through the execution of a *transControl* action (see Section 2.7.2). It is important that any such limitation or restriction is clearly brought to the attention of the reader.

Guidelines for the documentation of attributes for any abstraction layer are:

- For each interface, show all of the flows and attributes using the graphical notations (port by port).
- For each of those attributes or flows, provide additional documentation if needed.
- For all attributes that have semantics different from the defaults, describe the full semantics that apply to this interface.
- For all protocols, reference the relevant information in the protocol description section.
- For dynamic attributes (such as dynamic priority assignment), reference the selection criteria in the data types section.

If there are several attribute sets for one interface, define in operational groups. Name each group and repeat the above five steps.

2.7.1.7 Documentation Options

For the abstract interface layers with few ports per interface, the VC provider may choose to use the visual shorthand for all ports and port attributes in an earlier part of the VC documentation, specifically Interface: Overview and General Properties: Section 3.2.N.3.1. In this document configuration, the VC provider should cross-reference the earlier section from Behavioral - Port Attributes: Section 3.2.N.3.2 B1 $InterfaceName. The Behavioral – Port Attributes section then only needs to contain the textural documentation requirements.

Some port attributes described here may be viewed as channel attributes. These attributes include FIFOs, multi-rate channels, and pipelines. Channel attributes can be a property of the system-model or SoC integration not explicitly delivered with a VC. In these cases, the channel attributes may be assumed (required) properties of the environment, with these properties listed in Interface: Overview and General Properties: Section 3.2.N.3.1. The channel-like attributes are supplied with a specific interface description when the VC implementations/models supply some basic channel support in terms of buffering, pipelining, etc.

2.7.2 Behavioral – Port Transactions: Section 3.2.N.3.2 B2 $InterfaceName_

2.7.2.1 General Comments

Section 3.2.N.3.2 outlines the total possible set of supported transaction types broken down by applicability to specific abstraction layers. Note that in this section of documenting a VC interface, the types of transactions supported by each of the interface ports are listed. As such, the formats specified here are for describing the set of transaction types supported at each VC port. The specific transaction attributes were specified in the previous section of the deliverables document and are related to each transaction by association to a port. Any timing/ sequencing of transactions required for valid communication (such as, completion of a protocol) is described in the succeeding sections of this document.

Of the two types of behaviors that can be performed through a port: *t*ransactions and messages, only transactions may have a complex dataFormat associated with them. The reason for this is that an implied control semantic may only be associated with a transaction, not a message. Messages can only have an associated dataType.

Transactions are identified by the prefix trans. *M*essages are identified by the prefix mess. As messages are atomic, they are locally controlled actions and complete without regard to the state of the environment. Thus, a message cannot involve active two-way interaction with the environment as a requirement upon completion.

2.7.2.2 Layer 1.0 Interface Abstraction

- *transRead (Consumer, Initiator, or Responder*[default]*)* - Input Port only
- *transWrite (Producer, Initiator*[default], *or Responder)* - Output only
- *messSense (Responder)* - Sense an event. The *only* event type sensed at this level of abstraction is: *anyEvent*
- *messEmit (Initiator)* - Generate an event. Event type is: *anyEvent*

2.7.2.3 Layer 0.x Interface Abstraction

All Layer 1.0 transactions are OK, plus the following:

- *transOpenChannel (Initiator)* – May be used through a port for which there are multiple connective-channel options, as the operation to open a specific channel. *transOpenChannel* must complete prior to any further transactions occurring on the channel. This includes: *transWrite*, *transRead,* and *transSynchronize*. *transOpenChannel* automatically blocks the proceeding transactions from occurring until the channel is granted. When channel use is complete, the channel must be closed using the *transCloseChannel* operation. If there aren't multiple channels to select between, or if the channel selection has been broken out into separate control-operations at the current level of interface abstraction (e.g., *Req*, *Ack* and *Addressing* operations), it is not necessary to precede *transWrite*, *transRead* and *transSynchronize* operations with *transOpenChannel.* Format of *transOpenChannel* is as follows:

  t*ransOpenChannel (channelNameA*, *channelNameB*, …). These channel names must be supported by a system-model utilizing the component. *transOpenChannel* cannot be a message, as it must have an implied control semantic involving active two-way interaction with the environment.
- *transCloseChannel (Initiator)* – Used for closing of a channel opened with *transOpenChannel*. *transCloseChannel* cannot be a message, as it must have an implied control semantic involving active two-way interaction with the environment.
- *transRead* and *transWrite* – Are supported on uni and bi-directional ports.
- *transSynchronize (Initiator or Responder)* – Used as *c*ontrol for synchronization that is not necessarily dependent upon the transfer of data. This may consist of a rendezvous-type operation where the receiver may participate in the definition of the rendezvous point, or may simply consist of notification (e.g., a trigger signal such as a chip-select line). Any transaction of this type must be accompanied with an abstract behavioral description. *transSynchronize* cannot be a message, as it must have an implied control semantic involving active two-way interaction with the environment.
- *transReset (Initiator or Responder)* – This is *a* transaction to reset the internal state of the VC. Any transaction of this type must be accompanied with a description of the reset-state. Format of the transaction-type description is as follows:

  transReset(*resetStateA* = {*resetStateDescriptionA*},
  resetStateB = {*resetStateDescriptionB*}, …).

resetStateDescription may reference a description of reset-states accompanying the VC behavioral description. This cannot be a message, as it must have an implied control semantic involving active two-way interaction with the environment.

- *transControl (Initiator or Responder)* – This is a transaction to control the state of the protocol block. This may be used to switch the block between attribute operational groups. The control may imply changing the state of the protocol block to a new point in a given sequence, or changing the protocol block behavior to a different type. Any transaction of this type must be accompanied with an abstract description of the control-options available, or cross-reference to the attribute groups defined in the previous section. Format of the transaction is:

  transControl(*controlStateA* = {*controlDescriptionA*},
  controlStateB = {*controlDescriptionB*}, …).

  transControl cannot be a message, as it must have an implied control semantic involving active two-way interaction with the environment.

The following messages are also available at this abstraction level:

- *messRead (Consumer)* - Cannot have an implied control semantic associated with it.
- *messWrite (Producer)* - Cannot have an implied control semantic associated with it.
- *messSense (Responder)* - At these abstraction levels, the sense operation may be filtered. Allowable filters on the sensing operation are:
  - rising Edge
  - falling Edge
  - *anyEvent* - Can be applied to sensing on a port with a data type more complex than bit as well as for simple ports.
  - *specificDiff* – Works when sensing on a port with a data-type more complex than bit. In this case of *specificDiff,* the type of difference being sensed must be specifically identified. Description of *specificDiff* is given by describing initialState, finalState as a list of sensitivities.

    Format for the *messSense* description is:

    messSense(eventType, (initialState1, finalState1),
    (initialState2, finalState2), … /*for *specificDiff* */).
- *messEmit (Initiator)* – Event types as specified for messSense at this abstraction.

2.7.2.4 Layer 0.0 Interface Abstraction

Layer 0.0 is the most refined layer of interface abstraction. At this layer there are only four message types available. No complex (nonatomic) transactions are permitted. The messages may be defined as synchronous to a particular clock, or be completely asynchronous. If synchronous to a particular clock, the clock name must be noted along with the transaction name. Note that at this layer "Control" actions constitute events, while "Data" actions require persistence of signal values. The message types available are:

- messRead {Consumer; Initiator ONLY} – Data port only.
- messWrite {Producer; Initiator ONLY} - Data port only.
- messSense {Responder} - Control port only. Only event-types: risingEdge, fallingEdge, anyEvent.
- messReset {Initiator or Responder} – Untyped event used in a reset action.

Note*:* *messEmit* is not a message permitted at Layer 0.0. The reason for this is that the form of any generated signal at this level must be fully defined. The action of emitting a particular signal is achieved through the use of *messWrite*.

The VSIA standard does not restrict the mechanisms or syntax used to implement the above behaviors. However, it does require that the standard used be specified by the vendor and that the VSIA transactions listed above be related to the elements of that standard.

An example of an implementation of the interface transaction in terms of standard communication syntax follows. Consider the transaction sequence: (*transOpenChannel*, *transReadObject).* This is equivalent in the OMI Standard [3] to the registering of an *omiNetSensitivity* callback routine.

2.7.2.5 Behavioral Typing

The transactions (part of the port behavior) imply constraints upon the nets to which they are connected. Note that these must apply in conjunction with the attribute compatibility. Without defining operational sufficiency, the necessary properties which must be supported by at least one other port connected to the net (*Requirement on Connected Net*) and required to be absent from any other port connected to the net (**must not** *exist on Connected Net*) are shown in Table 9.

Table 9: Behavioral Typing by Transactions on Ports

Transaction on VC Port	Requirement on Connected Net	MUST NOT exist on Connected Net
transRead	transWrite	messWrite, messEmit, transSync, transReset, transControl
transWrite	transRead	messWrite, messEmit, transSync, transReset, transControl
transSync	transSync	Transaction or of any other type and messEmit. (messSense is OK.)
transReset	transReset	Transaction or of any other type and messEmit (messSense is OK)
transControl	transControl	Transaction or of any other type and messEmit (messSense is OK)
messRead	messWrite or transWrite	messEmit, transSync, transReset, transControl
messWrite	messRead or messSense	transRead, messEmit, transSync, transReset, transControl
messSense	No restriction	No restriction
messEmit	messSense	transRead, messEmit, transSync, transReset, transControl, messRead

2.7.2.6 Documentation Requirements

The following are requirements for the documentation.

- For each port, list the set of transactions and messages it supports.
- Each transaction and message associated with a port must be identified by the name: *$portName_$transType* or *$portName_$messType* respectively.
- For each transaction or message, if the data type is not specified by the port, the data type specific to each transaction and message must be specified. For each transaction, if there is an associated dataFormat, this must also be specified (see Section 2.3 Data Types). At Layer 0.0, the dataFormat can only consist of a dataValue field.
- If a transaction has a specific priority associated with it (priority over other transaction types on the same port), specify the set of transaction priorities.
- For each transaction, if there is a specific syntactic support for this transaction type (for examplem relationship to a specific proprietary transaction set, or breakdown of the behavior into an executable form), describe that support or a reference to it here.

Table 10 is an example structure for representing the basic transactions information. Detailed behavioral descriptions for each transaction type are assumed to follow this table unless cross-referenced to another document or section of this deliverables document (for example, overview and general properties for the layer, or an earlier port transaction section for a more abstract interface layer).

Table 10: Example Structure for Overview of Port Transactions

Port Name	Attributes	Transaction Class	Transaction Name	Data Type	Behavior
Data_Port_1	Blocking, Buffer[8]	transRead	PopData()	-- (See Port Defn)	Cross-reference
		transWrite	PushData()		Cross-reference
Data_Port_2	Buffer[1], Non-blocking	messRead	GetData()	vsi_bitvector(2) (VSIA SLD DTD)	Cross-reference

2.7.3 Behavioral – Inter-Layer Behavioral Mapping: Section 3.2.N.3.2 B3 $Interface_Name

2.7.3.1 General Discussion

In this section, the transactions defined at the more abstract level are mapped to transactions at this level. The static (name and type) mappings are defined here, while the dynamic (timing and sequencing) mappings are defined in the next section. In general, for mappings between levels of abstraction, the set of Requirements/Assumptions from the more abstract interface layers are identified as sets of constraints upon the interface behavior at the current layer. This mapping may support many-to-one, one-to-many, and many-to-many associations. However, the former two are greatly preferred to the latter since they enable a much more understandable structure to be given to a multi-level interface. As explained below, many-to-many mappings are usually complex and difficult to document clearly. Thus, it is recommended that these be simplified by factoring out an intermediate layer.

The argument against many-to-many:

> It is possible to support an inter-layer behavioral association that maps the behavior of multiple ports at one layer of abstraction to multiple ports at a more refined layer of abstraction. An example of this may be two output (*transWrite*) ports at Layer 1.0 which are actually mapped into a shared set of ports at Layer 0.x. At Layer 0.x, the different write operations may be identified through a specific sequencing operation which is implied by the protocol. More specifically, at Layer 1.0 we have *transWrite* (*blocking*) at *PortA* and *transWrite* (*blocking*) at *PortB* map into a Layer 0.x where sequenced *transWrite* (*nonblocking*) transactions on *DataPort* along with sequenced *Req/Ack* control performs the operation. In moving from Layer 1.0 to Layer 0.x interface abstractions, two distinct mechanisms must be described. On one hand, the more abstract sequencing of the *transWrite* operations inherited from Layer 1.0 must be explained. However, the sequencing of the *DataPort* and *Req/Ack* transactions must also be explained – a sequence required for generic write operations. In more complex cases, these many-to-many mappings are very difficult to describe clearly. One possibility to consider, when a many-to-many mapping seems unavoidable, is whether the set of transactions at the higher level could have been better described as a single transaction.
>
> It is recommended, wherever practical, that IP Providers determine the layering of their interface description in such a way that only single aspects of communication abstraction are revealed in moving between two adjacent layers. In the case of the example provided here, the IP provider should include an intermediary interface layer (for example, Layer 0.y) which supports a communication abstraction between Layer 1.0 and Layer 0.x. The purpose of this intermediary layer is to clearly delineate the sequencing applied to the two Layer 1.0 *transWrite* transactions when mapped to a single port supporting *blocking* write on Layer 0.y. Layer 0.y to Layer 0.x need then only show the mechanism for translating this blocking write into a *Req/Ack* control sequence and a nonblocking *transWrite* on *DataPort.*

Mappings which are one-to-many or many-to-one will need a protocol specification to define how the many combine to meet the needs of the one. Simple one-to-one mappings will not require this, but may still require a data mapping specification if the data types of the transactions are different.

2.7.3.2 Documentation Requirements

For each transaction listed at the layer of abstraction one higher (Layer 0.x) than the current layer (Layer 0.y):

> Proceed as per the interlayer associations indicated: List the abstract transaction/ports, and list the set of transactions/ports associated with each of these at the current layer. A reference to the defining protocol in the next section or to a data type mapping specification should be included if either of these exists. The type mapping specifications should be in the same format as in Section 2.6.3.4 on port data type mappings. Also, as in Section 2.6.3.2 on port mappings, a table can be used to structure the lists.

2.7.4 Behavioral – Protocol Description: Section 3.2.N.3.2 B4 $InterfaceName

In Protocol Description: Section 3.2.N.3.2, the interaction of the VC with the external world in terms of transaction timing and sequencing is described. Also the general constraints carried over from the next more abstract layer, and not yet met by earlier simpler mappings, will be met by a mapping to the transaction behaviors at this level. These constraints may include:

- *Completion Constraints* – Time in which a behavior must complete after commencing.
- *Wait Constraints* – Time in which a behavior must commence.
- *Sequencing Constraints* – Implied order of communication tasks.
- *Other* – The constraint type must be comprehensively and quantifiably described.

Note that the protocol description section is not required at Layer 1.0, where it can be omitted or, optionally, a description of important temporal and/or functional requirements and their mappings to the VC interface can be inserted.

2.7.4.1 General Discussion

An explicit description of the behavior, for example in terms of finite state machines (FSMs) and so forth, should include the consideration of:

- Sequencing/Concurrency Requirements and Assumptions
- Sequencing/Concurrency Properties
- Temporal Interpretation

These considerations may influence the description given in the protocol description section either explicitly in the text or implicitly as a result of the formalism chosen to capture the behavior. For example, an FSM description would need an explicit specification of the temporal interpretation, whereas this would be intrinsic to a waveform diagram. In another example, the sequencing/concurrency properties may only be useful for VCIs with special repetitive guaranteed patterns of behavior.

The essential properties specified in this section should be the patterns of communication allowed by the specified transactions.

- *Sequencing/Concurrency Requirements and Assumption* implies documentation of any temporary or sequence-related behaviors which must be supported by the environment in which the VC is embedded or by the interface layers more refined than the current abstraction. This may also include the specification of constraints on temporarily related interactions. These relationships may apply to transactions through single or multiple ports. As a single port example, a *transReadObject* request through port X may need to complete prior to the commencement of another *transReadObject* request through that same port. A multi-port constraint example may require that a *transReadObject* request to port X cannot coincide with a *transWriteObject* request on port Y.

- *Sequencing/Concurrency Properties* are sequencing of transactions through single or multiple ports, which will always occur through the interface layer. These properties are guaranteed by the behavior of the VC, or the behavior of the interface protocol at this abstraction layer, and may be used to optimize the more refine protocol layers of the interface. *Sequencing/Concurrency Properties* may apply to the identification of data objects passed through an interface and decomposed at this interface layer. For example, at this interface layer we may identify that for all n the *Im* part of a complex number is the $(2n+1)^{th}$ object passed through the interface, the $(2n)^{th}$ object passed being the associated *Re* part.
- *Temporal Interpretation* describes how the VC communicates its version of time with the external environment. Time may be communicated as the triple: (#_of_events, type_of_event, error). This is a requirement if the Layer 1.0 interface is to support the concept of *timing windows*. (That is, the indeterminacy of time handled in one model when compared with the precision of another.) As an example, consider the coupling of a discrete-event model to a cycle-based model. The abstraction of time in the cycle-based simulation does not necessarily imply that the relative time in the discrete-event world is exactly *number_of_clock_periods*clock_period*. Such an approximation does not consider the possibility of clock drift. Drift should be modeled by a window of indeterminacy. Timestamps for objects sent across the Layer 1.0 interface would then be: (#_of_events, type_of_event, error). *Temporal Interpretation* specifies how the VC resolves the concept of timing indeterminacy handed to it. That is, *Temporal Interpretation* identifies whether several possible outcome scenarios were executed, an average condition was calculated, or a random time in the interval was selected.

  An example of object timestamps is: (#_of_events, type_of_event, error) = (5 events, Clock_Positive_Edge, 0.5*Clock_Period).

  The timing/sequencing of transactions in the specification of a protocol description may assume any method agreed to by the VC provider and VC integrator. However, it is recommended that the method chosen should include the capabilities of the suggested documentation method described below. Additionally, the protocol should be specified both ways. The documentation standard following would be a standard way of describing the protocol, and the additional format would be a computer language that is more easily exchanged, simulated, and verified.

  Examples of some methods are:

  - FSM, Extended FSM, CFSM, Timed FSM
  - StateCharts
  - VHDL+, SDL, Esterel, Lotos
  - Waveform Diagram
  - Timing Diagram Markup Language
  - Message Sequence Charts
  - Petri Nets (usually for asynchronous behavior)
  - Interface Protocol Description Languages: e.g., Owl [4]

  The protocol description must be provided in a manner that clearly identifies the behaviors of the attributes. Each VC provider is responsible for documenting an interface and adopting a formalism, accompanying it with a further document of explanation or a reference in which the syntax is comprehensively described.

  In the case of control registers within the VC interface, it is critical that they be described in this part of the document. In particular, these registers must be related to the setting and maintaining of state as used in the execution of the protocol. (For example, relate the existence and values of the control registers to the states used within the FSM description.)

2.7.4.2 Documentation Requirements

- For each named protocol in earlier sections (Section 2.7.1.6 Port Attributes and Section 2.7.3.2 Inter-layer Behavior), provide a specification of the protocol which defines the attributes discussed above. This specification should be in the format described below but may be specified in an equivalent format as well.
- The documentation format suggested for the specification of the protocol is a discrete temporal graph, with time along the horizontal or X axis, and the vertical or Y axis being a set of unordered, but connected, activities or resources. The graph of each protocol should not be regarded as an imperative behavior but rather as a declared pattern in time.

- Each discrete element of the graph represents a transaction or a temporal constraint. Connecting these elements are control arcs that decide which element occurs next in the pattern. This graph can be drawn graphically, but a simpler – and much faster to capture – method is to use a table. In the table, the common control arc representing sequence can be implicitly captured by column order, and other control arcs can be entered using special text and labels. The maintaining of state, as indicated by temporal graphs or sequencing diagrams, should be mapped to the physical instantiation of the state within the interface object when possible (for example, control registers, and so forth).
- Additional textual documentation may show that the sequencing/concurrency requirements and assumptions passed down from the more abstract interface-layer are satisfied by the behavior and within the set of requirements/assumptions made.
- If there is a difference in temporal interpretation between this Interface layer and the more abstract layer, the conversion between the two should be documented here, either textually or as a data translation.

Table 11 is an example representing protocol attributes. Each row is implicitly serialized by default (from the protocol). All the transactions are on port 1, but that may be made explicit in the table. Control arc cells are not shaded. The list of transactions in the protocol must conform to the list given for this protocol in Section 2.7.3.2.

This example is not meant to make sense as a real example, but just to give the flavor of protocol specification.

Table 11: Protocol Specification

Protocol One_A: time increasing somehow this way >>>

<table>
<tr><td rowspan="3">C=trans Open Channel</td><td rowspan="3">W=trans Write</td><td>T1(T1min to T1max) = Delay</td><td colspan="4">C=transCloseChannel
R=trans   L: Loop R   A[R}=transR   End</td><td rowspan="3">E:R=trans Write</td></tr>
<tr><td>T2(T2min to T2max) = Delay
When V = Port2.messWrite then exception E</td><td>Read</td><td>to 1</td><td>ead</td><td>Loop L</td></tr>
<tr><td colspan="5"></td></tr>
</table>

2.8 Behavior/Interface Association: Section 3.2.N.4

The behavior-interface association enables actual behavior models, when supplied, to be mapped to the interface at the level of abstraction at which the behavior is written.

2.8.1 General Comments

Since the only mandatory behaviors are at level 1.0, this section is only mandatory for Layer 1.0. For all other layers, this is Conditionally Mandatory. The condition is as follows: if a behavioral model *is* offered at this layer of abstraction, then the connection of the interface to this block must be shown.

All assignments must be type and behavior compatible. That is, the behavior of mapped IO variables, ports, or actions within the behavior must match the specified properties, attributes, and constraints of the port in the interface.

2.8.2 Documentation Requirements:

- For each specified IO variable or port within the behavior, assign it to a port at this level of abstraction
- For each specified IO action from within the behavior, assign it to a port at this level of abstraction

3. Known Issues

The following is a set of known issues, which are opened by the standard. These issues are to be closed in future versions.

Specification of suitable standards for representation of timing/sequencing aspects of the protocols.

4. References

The following are recommended references:

On-Chip Bus Attributes – Version 1.0, On-Chip Bus Development Working Group, Virtual Socket Interface Alliance, 1998.

J. Rawson, A. Sangiovanni-Vincentelli., *Interface Based Design*, Proc. of Design Automation Conf., June 1997.

IEEE, Open Model Interface Documentation v3.0, IEEE P1499 Technical Committee, December 1997.

K. Suzuki, et al, *Owl: An Interface Description Language for IP Reuse*, in Proceedings of the IEEE 1999 Custom Integrated Circuits Conference, pp. 403-406, May 1999.

Ptolemy || Heterogeneous *Concurrent Modeling and Design in Java v0.1.1*, ERL Technical Report, University of California, Berkeley, February 1999.

J.P. Calvez, *A System Specification Model and Method*, in Current Issues In Electronic Modeling,

Volume 4: *High-Level System Modeling: Specification and Design Methodologies*, R. Waxman, J.M. Bergé, O. Levia, J. Rouillard (eds), Kluwer Academic Publishers, 1996, pp 1-54.

5. Appendices

The following appendices are provided as further motivation and explanation of the material contained within this standard. The purpose for inclusion of these appendices is as follows:

A: An Example System-on-Chip Interface-Hierarchy

Provides a system-view of the design process, and explores the concept of iterative refinement. Through this view, we provide some basic motivation and a simple example of how a designer may consider the "layering" of the system-level interface

B: Layer 1.0 to Layer 0.0 Behavioral Association

This example shows the simple but detailed behavioral association between a Layer 1.0 and Layer 0.0 description for a single component interface. The example does not follow through the details of this standard in terms of structuring and completeness of information. However, it provides an immediate conceptual depiction of the interface abstraction concept as applied to a single VC.

C: Interface-Layering Documentation Example

Contains the System-Level Interface Behavioral documentation for an example system-level Virtual Component (VC) called Constant Multiplier (CMULT). CMULT was developed in a VSI Pilot Project performed by IMEC, Belgium. Although this example was authored within the Ocapi system-development environment, the documentation shows how description of the VC can be made environment-neutral.

The CMULT design is a constant multiplication block, intended for implementation in hardware. The function performed is out = in * P, with P a programmable parameter. The block is described at four different layers of abstraction, starting with a dataflow C++ model, and ending at a cycle-true RT-VHDL model.

The example shows correct naming and numbering schemes for VC documentation. The example clearly demonstrates a technique for the introduction of clocked transactions when moving down from a Layer 1.0 model to a Layer 0.0 abstraction

A. An Example System-on-Chip Interface Hierarchy

The following example is more specific to system-on-chip. This is closer to the expected direct application of the *System-Level Interface Behavioral Documentation Standard.* Note that for reasons of complexity, the full resolution of all system interfaces from Layer 1.0 down to Layer 0.0 is not shown. However, this example also illustrates the concept of iterative interface refinement without the need to refine all aspects of system communication.

The black blocks are elements of the interface hierarchy associated with each of the virtual components. Note that for this example, each point of connection within a channel has an associated single transaction. This is not a restriction of the standard, but a simplification for this example. (*Data* and *Address* are equivalent to the transactions *Write Data/Address.*)

Note that the suggested dominant properties of the layers are again an example and need not be followed as an SLD layering principle.

In Figure A1, a basic conceptual model of communication between task A and task B is shown. In this case, the transactions implicitly require a large amount of control definition, as with the concept of blocking and the concept of queue management.

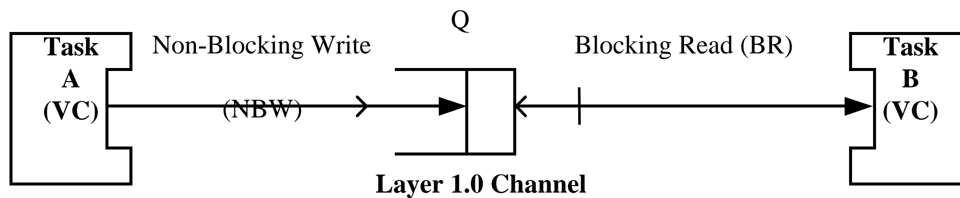


Figure A1: Top-Level Conceptual Communications Between Tasks

In Figure A2, the elements of protocol hierarchy associated with each of task A and task B are broken out into their generic processes. The abstract notion of a queue is identified to be composed of a memory block and control unit, both of which are VCs. To manage the notion of such a queue, there must be the definition of actual relative addresses (not absolute, as we are not yet system implementation specific) and control associated with the notion of the blocking read – in this case, a simple Request/Acknowledge. Note that this example illustrates the concept of protocol-block state and sequential association of transactions. In this case, the relative address for the data being read by task B would not be sent to the memory block (that is, the read request is gated) until an acknowledge (*Data Present Acknowledge*) to the data request is received by the protocol block.

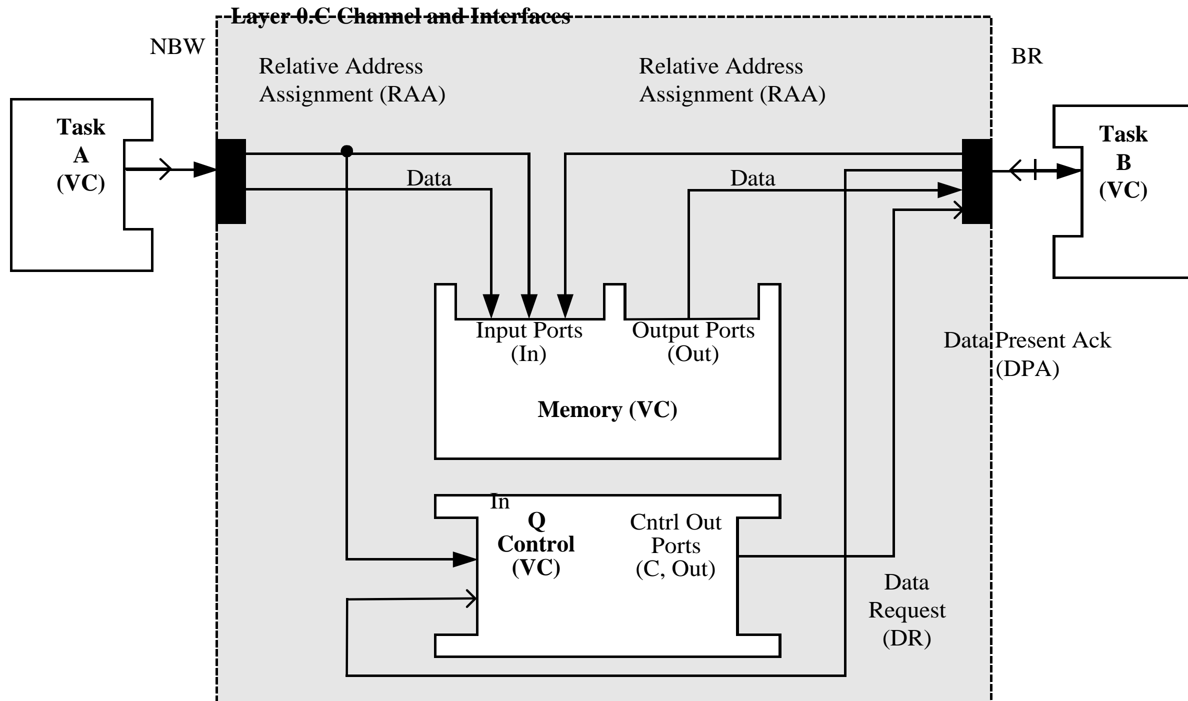


Figure A2: First Level of Detailing in the Channel - Identifying Basic Control

In the next most refined layer of interface specification, as shown in Figure A3, the identification of the need for communication with general system management is realized. In this case, the new virtual components identified, the driver, and the operating system (OS), are called upon. The OS is responsible for managing the library of drivers, in this case calling the appropriate driver associated with task A when that task attempts to write to memory. The driver itself (abstractly) is responsible for translating the relative address generated by task A into an absolute system address and manages access to the shared memory block. (In the interests of simplicity, Figure A3 is a partial interface refinement for the complete set of interfaces in the system.)

Layer 0.C Channel and Interfaces
Layer 0.B Channel / Interfaces
Memory Write Req/Ack
(MWRA)
NBW
RAA
RAA
Driver
(VC)
(RAA)
BR
Task
A
(VC)
Data
Data
Data
Data
Task
B
(VC)
Load
Driver
(LD)
AA
Absolute
System
Address
(ASA)
DR
In
Out
Memory (VC)
OS
(VC)
OS Call
(OSC)
In
C. Out
Q
Control
(VC)
RAA
DPA

Figure A3: Second Level of Detailing in the Channel - Identifying System-Specific Control

The models of Figures A1 through A3 all specify the concept of inter-block communication without considering the concept of shared communication resources, such as a bus. That is, up to this point, all communication is *dedicated.*

In this example, there are several possible sources for the use of communication objects not dedicated to serving the single needs of task A communicating with task B. One such shared element that has already been resolved out from the Layer 1.0 channel is the operating system. The interfaces to this unit may be resolved out to reveal a protocol, which handles multiple requests.

Another, perhaps more commonly recognized element of shared communication is the on-chip bus (OCB). There has been significant effort within the OCB group of the VSIA to resolve an abstraction-layering principle for such a system bus. This bus-specific layering contained within the concept of the OCB virtual component interface (VCI) hooks directly into the layering principles being proposed by the interface group of the SLD Development Working Group.

The transaction set being proposed by the OCB Group will be a subset of all possible abstract transactions supported in the interface group standard. Furthermore, the description of protocol layering specific to buses will be compatible with the layering-description techniques used for the general system-interface hierarchy. Conceptually, Figure A4 illustrates how the OCB model will connect into the system-level interface hierarchy.

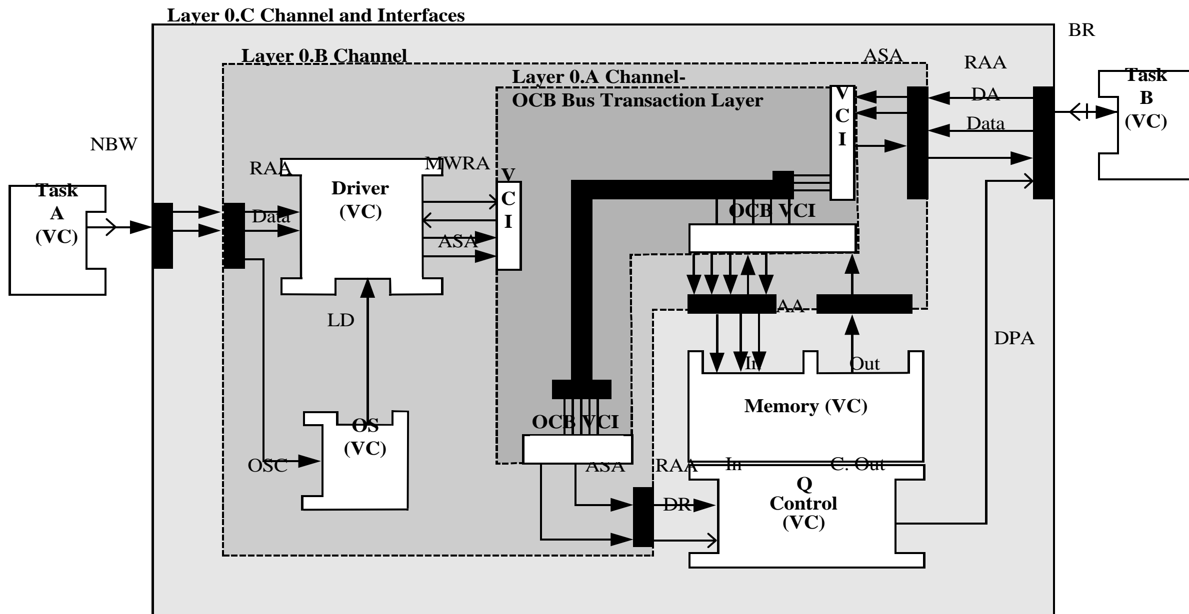


Figure A4: Third Level of Detailing in the Channel - Shared Communication Resouces, Specifically the OCB

For this example, a final level of interface abstraction is shown in Figure A5. In this case, we are just considering the VC task A from its abstract, Layer 1.0 interface to its Layer 0.0 protocol specific or implementation specific, interface. This layer, as shown, fits between task A and the virtual components with which it shares an immediate connection – namely the operating system and the driver.

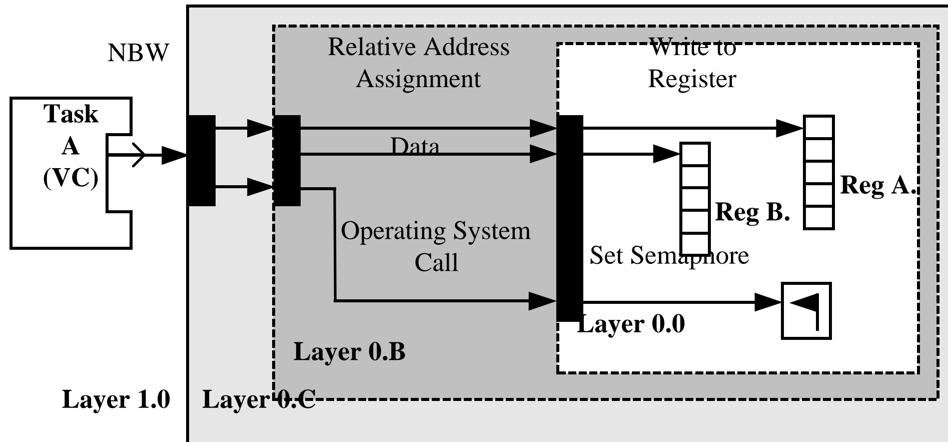
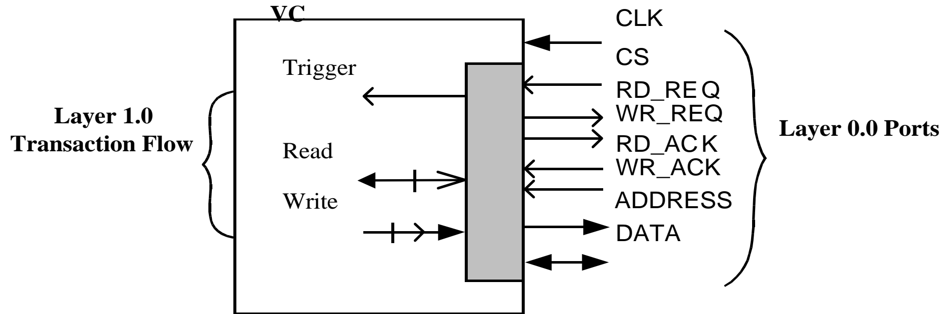


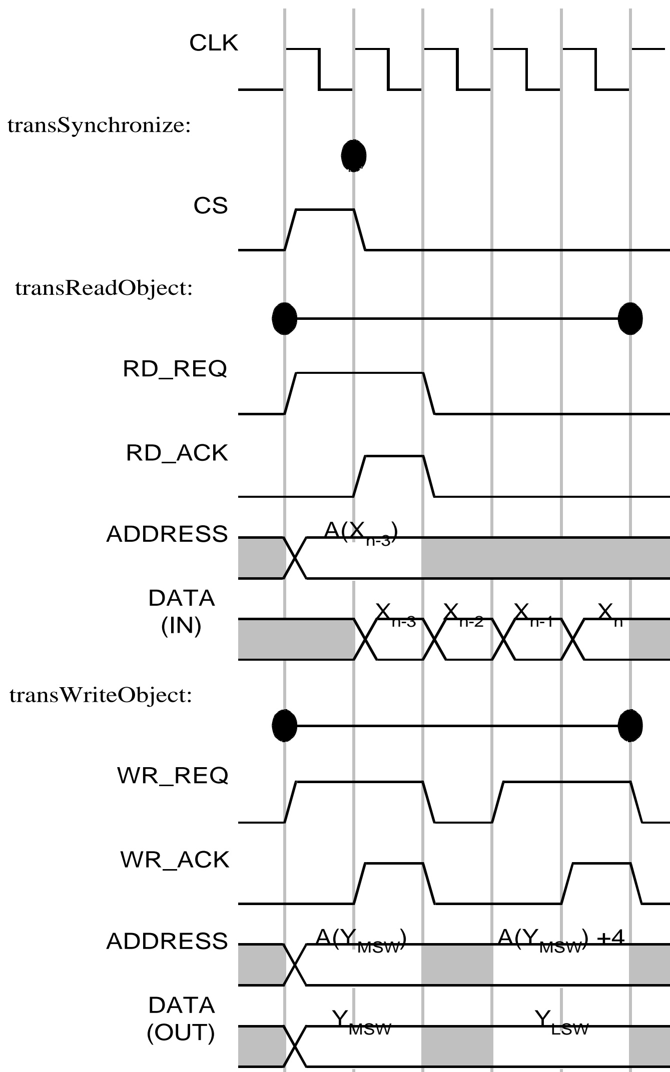
Figure A5: Final Level of Interface Detailing in the Channel - Protocol or Implementation

B. Layer 1.0 to Layer 0.0 Behavioral Association

Structural View:



Behavioral View:



Functional Description:

All IO transactions are synchronized to a single clock.

Trigger occurs when the chip select line is set for a single clock cycle indicating that all input data is ready. This will initiate VC operation.

Read operations require burst-mode transfer of 4 data long-words (32 bits). Read request is indicated when the VC sets the RD_REQ line. The RD_ACK line set indicates that data at the input port is valid for read. Burst mode data is sent as indicated in the timing diagram.

Write transactions are composed of two long-word (32-bit) writes. Each long-word write requires a full handshake initiated with the WR_REQ line being set, and completed when a WR_ACK signal is returned. The most significant long-word of the output Y is sent first, followed by the least significant word. Base address for the output write is increased by 6 bytes for each time the VC operation is initiated.

C. Interface-Layering Documentation Example

Constant Multiplier VC:
System-Level Interface Behavioral Documentation

1 User Guide

2 Model Implementation

The constant multiplier VC is an example VC to illustrate the use of the *System-Level Interface Behavioral Documentation Standard*. This VC can be retrieved as a software package from http://www.imec.be/ocapi/Welcome.html. Upon unpacking the software, the following structure is created:

CMULT/

doc/Interface documentation

test/Testbenches to drive VC

vc/Virtual component source at different abstraction levels

Throughout the documentation, references to files in these directories and identifiers in those files will be indicated with underline formatting, for example class BlockingWrite, file vc/df2rt.h.

3 Technical Attributes

3.1 Layering Principle

The CMULT VC is described at three different abstraction levels:

a. A functional level that uses dataflow semantics.

b. An architecture level that uses dataflow semantics.

c. An architecture level that uses synchronous RT semantics.

During design refinement (when moving from a to c), an interface-based design mechanism is used. This strategy first translates each of the ports at dataflow level into an equivalent RT-level port. Next, the behavior of the block itself is converted. Block behavior conversion includes:

- Refinement of operations to clock cycles.
- Refinement of floating point operations to fixed point ops.

This way, a set of different simulations is obtained, at different levels of abstraction. The following layers are defined within the model.

Layer	Name	Section	Beh	Intf	Impl
1.0	Dataflow	3.2.0	Y	Y	N
0.2	DF-interface	3.2.1	Y	Y	N
0.1	RT-interface	3.2.2	Y	Y	N
0.0	RT	3.2.3	Y	Y	Y

Where:

Beh = An Executable Model (C++) is included.

Intf = An Executable Model (C++) of the Interfaces is included.

Impl = VHDL Implementation is included.

The correspondence between the abstraction levels and the layers is as follows: (a) corresponds to layer 1.0, (b) to layer 0.2, and (c) to layers 0.1 and 0.0. All of the levels are captured in C++. The lowest level (0.0) can be converted to VHDL automatically in OCAPI. A brief description of the purpose of the layers follows:

Layer 1.0 is a high level, functional model intended for inclusion in high level simulation environments. The functionality is expressed in data-flow semantics. In this layer, the interface behavior is expressed directly in terms of the primitives present in the data-flow simulation environment.

Layer 0.2 is a functional model with explicit modeling of the behavior of each interface (protocol). Layer 0.2 uses data-flow semantics for expression of all behaviors. Therefore, this layer can still be used in combination with high level environments.

Layer 0.1 is a functional model with explicit modeling of the behavior of each interface (protocol). In contrast with layer 0.2, the interface behavior is expressed in terms of an architecture data model (FSMD) rather then a data-flow data model.

Layer 0.0 is an architecture model that combines the function of the protocols and the block-local behavior in one schedule. This layer is expressed using the semantics of the architecture data model and also in VHDL semantics.

The relationship between the different files and the documentation at different layers is expressed in Table C1. This will allow the reader to navigate through the VC code.

Table C1: File Documentation Relationships

	Layer 1.0	Layer 0.2	Layer 0.1	Layer 0.0
VC Description				
vc/cmult.h	***	***	***	
~~vc/df2rt.h~~		***		
~~vc/df2rt_~~clock.h			***	
~~vc/cmultrt.h~~				***
~~**Testbench**~~				
test/level1_0.cxx	***			
~~test/level0_2.cxx~~		***		
~~test/level0_1.cxx~~			***	
~~test/level0_0.cxx~~				***

3.2 Layer Documentation

3.2.0 Layer 1.0

3.2.0.1 Data Types

Datum	`dfix_double`
Datavalue	Double (-1.79e+308..1.79e+308)
Datasize	8 bytes

3.2.0.2Internal VC Behavioral Description

The internal behavioral description is given by:

O = P * I where:

- The value of `I` is read from an external interface `dsp.`
- The value of `P` is read from an internal state var `state_P.`
- The value of `O` is written to an external interface `dsp.`
- The state var `state_P` is updated through an external interface `parm.`

The internal behavioral description is included in vc/cmult.h. The expected system communications semantics are data-flow: data-flow actors that communicate through data-flow queues. The modeling is assumed to proceed with computation quantums (i.e., the iterative behavior that is expressed by a data-flow actor is atomic). The behavior cmult derives from a `base` type, driven by the system scheduler. The behavior of cmult is described in the run() function. First, the input parameter P is read (if available). Next, a firing rule test is done in input I. If successful, I is read, multiplied by the current parameter value and sent to the output. The reader should note that the behavior is not a pure dataflow block with a single firing rule. The update of parameter `lastP` and the multiplication `lastP * I` can proceed independently.

3.2.0.3 Interfaces

3.2.0.3.1 Overview and General Properties

The VC contains the following interfaces:

Name	Description
`Dsp`	Reads input values to multiply, and provides results.
`Parm`	Reads parameter values for multiplication

Figure C1 illustrates how the interfaces are related to ports. Also, the transaction types at the interfaces are expressed symbolically. All datums of data nature flowing over either `dsp` or `parm` are of type `dfix_double`.

The following transaction types exist:

- `dsp` uses a blocking read. Tied to this blocking read is a nonblocking write; ie. for each input value that is read on port I, an output value is produced on port O.
- `parm` uses a non-blocking read.

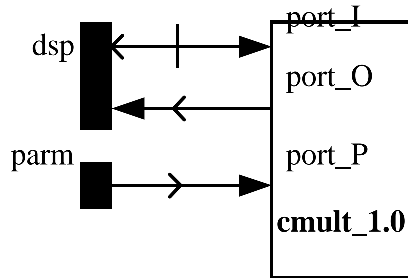


Figure C1: Layer 1.0 External View

3.2.0.3.2 Interface-specific Description dsp

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
port_I	Cons	`dfix_double`	Reads input I
port_O	Prod	`dfix_double`	Writes output O

BEHAVIORAL – PORT ATTRIBUTES

See Figure C1.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
port_I	transRead(Blocking)
port_O	transWrite(NonBlocking)

3.2.0.3.3 Interface-Specific Description parm

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
port_P	Cons	dfix_double	Reads parameter P

BEHAVIORAL – PORT ATTRIBUTES

See Figure C1.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
port_P	transRead(Non-Blocking)

3.2.0.4 Behavior/ Interface Association

The following relations exist between the description in vc/cmult.h and the interface definitions of Section 3.2.0.3.

Interface/Port/Action	C++
port_I, port_P, port_O	I, P, O of cmult constructor parameter
transRead(Blocking)	Test on I.getSize() followed by I.get()
transWrite(Non-Blocking)	Unconditional O.put()
transRead(Non-Blocking)	Test on P.getSize() and optional P.get()

3.2.1 Layer 0.2

3.2.1.1 Data Types

3.2.1.1.1 General Types

3.2.1.1.2 Transport Types

Datum	Dfix_bool
Datavalue	Bool (false, true)
Datasize	1 bit

3.2.1.2 Internal VC Behavioral Description

The Layer 0.2 of the CMULT VC is a wrapping of the behavior of the 1.0 Layer with RT-level interfaces. The functional behavior is identical. Layer 0.2 still relies on dataflow simulation semantics to express the functionality of the different ports at RT-level. Since dataflow semantics are untimed, they cannot express the timing relationship between the different ports. Layer 0.2 merely introduces the functionality of these RT-level ports (the interface signals, the type of data values generated on them and read from them, and the different tests that are used in protocol sequencing).

Layer 0.2 is implemented in the simulation environment as a 'peel' around layer 1.0. Only the interface refinements are designed; the internals stay identical. Internal operation is explained in Section 3.2.0.2 and listed in vc/cmult.h. The interface refinements are explained in Section 3.2.1.3.1 and listed in vc/df2rt.h.

3.2.1.3 Interfaces

3.2.1.3.1 Overview and General Properties

The VC contains the following interfaces:

Name	Description
Inp	Reads input values to multiply
Out	Produces resulting multiplicant
Parmrt	Reads parameter values for multiplication

All datums of data nature flowing over iinp, parmrt, or out are of type dfix_double. All datums of control nature (flowing through _req and _ack ports) are of type dfix_bool.

The following transaction types exist:

- inp uses a blocking read on port_I. Through the control ports port_I_req and port_I_ack, signalling is done such that the complete interface inp works like a blocking read.
- parmrt uses a nonblocking read on port_P. All data appearing on this port_P is consumed when a control event on port_P_ack indicates that consumption is desired.
- out uses a nonblocking write. Data is produced in port_O, and a control signal is generated in port_O_req to indicate the presence of this data.

In Figure C2, an external view on Layer 0.2 is given. The figure identifies the ports related to each interface and the behavior of each port.

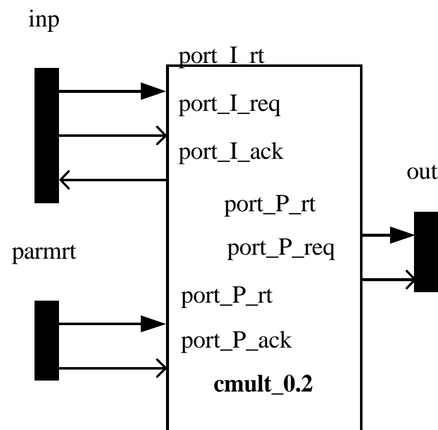


Figure C2: Layer 0.2 External View

Layer 0.2 actually encapsulates layer 1.0 by extending the functionality of the interfaces only (and keeping the functional core identical). A graphical representation of this is shown in Figure C3. Such a view of the internals is of course not unique. In this case, the internal view also corresponds to the actual implementation in C++.

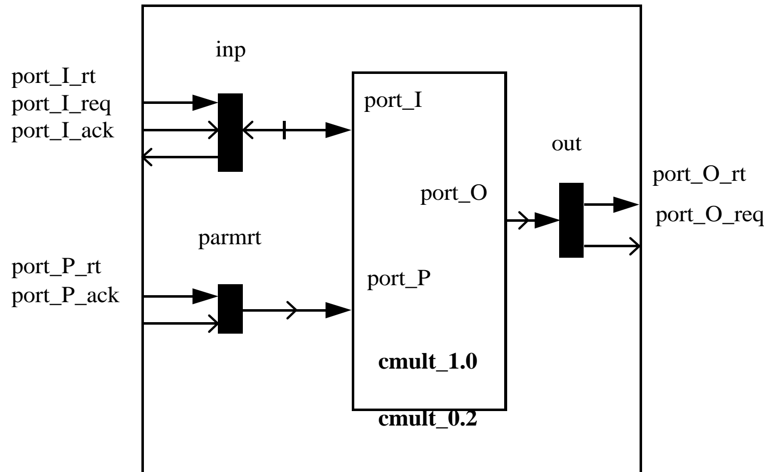


Figure C3: Layer 0.2 Internal View

The code in vc/df2rt.h describes how the interface blocks of Figure C3 are actually designed. The correspondence of this figure and the class definitions are as follows:

Class	Description
`Abstractproto`	Astract protocol block: This protocol block creates acknowledge and request ports and relates these to an input and output interface. Derived classes will then work out the signalling that takes place on the acknowledge and request ports when information flows from an input port to an output port.
`BlockingRW`	Figure C3, Interface inp: This class implements blocking semantics. When information is to flow, all request signals will have to be responded with acknowledge signals.
`NonBlockingRead`	Figure C3, Interface parmrt: This class implements NonBlockingRead semantics. These semantics do not use any request signaling. When an acknowledge signal occurs, data transport proceeds.
`NonBlockingWrite`	Figure C3, Interface out: This class implements NonBlockingWrite semantics. These semantics do not use any acknowledge signaling. When an request signal occurs, data transport proceeds.

These classes can be used to implement the different protocols that are running on the interfaces. The classes are executable blocks, and use data-flow semantics. They will also be described as abstract VSI transactions (like messSense, transRead etc.) in the documentation below. Consequently, a mapping was chosen from data-flow executable semantics to abstract VSI transactions. The mapping will be explained throughout the documentation by indicating the relation between the abstract description of a protocol and the concrete implementation in C++.

3.2.1.3.2 Interface-specific Description `inp`

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
port_I_rt	Cons	dfix_double	Reads Input I
port_I_req	Prod	dfix_bool	Generates request signal
port_I_ack	Cons	dfix_bool	Reads acknowledge signal

STRUCTURAL – INTER-LAYER STATIC MAPPING

Layer 1.0 port	Layer 0.2 Mapping
port_I	port_I_rt
	port_I_req
	port_I_ack

BEHAVIORAL – PORT ATTRIBUTES

See Figure C2.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
port_I_rt	transRead (initiator)
port_I_req	messEmit (anyEvent)
port_I_ack	messSense (anyEvent)

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

See Figure C3.

BEHAVIORAL – PROTOCOL DESCRIPTION

The Layer 0.2 protocol expresses a transRead(Blocking) transaction in terms of signaling on two control signals (port_I_req and port_I_ack).

```
port_I: transRead(Blocking) {
  port_I_req: messEmit(1)
  again:
  IF (port_I_ack: messSense())
   port_I_req: messEmit(0)
   port_I_rt: transRead
  ELSE
   port_I_req: messEmit(1)
   GOTO(again);
  }
```

Considering the code in vc/df2rt.h, this protocol can be found in the run() method of the BlockingRW class. The abstract calls messEmit() and messSense() map to the following concrete dataflow behavior:

Protocol Operation	C++ Implementation
`transRead`	Proceed with the read transaction. In the dataflow description, this will move a token from the input port to the output port: `out.put(in.get())`
messEmit(x)	Put a token of value x on the interconnect. `ReqPort << dfix(1)`
messSense()	Read a token from the interconnect and test its' value. If there is no token present, consider the event absent. e.g.: `int ackval = 0;` `if (ackPort.getSize()>0) //token present` `ackval = (int) ackPort.get().Val();`

The BlockingRW class in Section 3.2.1.4 also contains one extra aspect not present in the abstract description given above., T and that is, the following t. The BlockingRW class is meant to be be integrated into a dataflow simulation environment. Consequently, it also contains a condition that indicates when the transRead(blocking) transaction will proceed. This condition is the traditional dataflow firing rule. (The check for tokens upon the input signal in Section 3.2.1.4.).

3.2.1.3.3 Interface-Specific Description `parmrt`

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
`port_P_rt`	Cons	`dfix_double`	Reads parameter P
`port_P_ack`	Prod	`dfix_bool`	The presence of an acknowl-edge signal indicates reading of P

STRUCTURAL –INTER-LAYER STATIC MAPPING

Layer 1.0 Port	Layer 0.2 Mapping
`port_P`	port_P_rt
	port P ack

BEHAVIORAL – PORT ATTRIBUTES

See Figure C2.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
`port_P_rt`	transRead(Non-Blocking)
`port_P_ack`	messSense()

Behavioral – Inter-Layer Behavioral Mapping

See Figure C3.

Behavioral – Protocol Description

```
port_P: transRead(Non-Blocking) {
 IF (port_P_ack: messSense())
  port_P_rt: transRead
}
```

In the protocol descriptions included in vc/df2rt.h, this protocol can be found in the run() method of the NonBlockingRead class. In the dataflow implementation, the messSense() and transRead are actually quite symmetric. Whenever a token on the ack port is present, it will be read in the variable ackval. When this token indicates the presence of the event (ackval is nonzero), and there is also a token on the data input (in), then the transaction will proceed.

There is also a mapping from dataflow execution semantics to the abstract transRead(Non-Blocking). In the concrete description in vc/df2rt.h, the transRead(Non-Blocking) will be initiated whenever there is a token to transfer on the data input in.

3.2.1.3.4 Interface-Specific Description `out`

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
`port_O_rt`	Prod	`dfix_double`	Writes result O
`port_O_req`	Prod	`dfix_bool`	Generates a request signal whenever there is an output O to write

STRUCTURAL – INTER-LAYER STATIC MAPPING

Layer 1.0 port	Layer 0.2 Mapping
`port_O`	`port_O_rt` `port_O_req`

BEHAVIORAL – PORT ATTRIBUTES

See Figure C2.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
`port_O_rt`	transWrite(Non-Blocking)
`port_O_req`	messEmit(anyEvent)

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

See Figure C3.

BEHAVIORAL – PROTOCOL DESCRIPTION

```
port_O: transWrite(Non-Blocking) {
  port_O_ack: messEmit(1)
  port_O_rt: transWrite
}
```

This protocol can be found in (vc/df2rt.h), the run() method of the NonBlockingWrite class. messEmit() is implemented by putting tokens on the interconnect, while the transWrite transaction actually tranfers data from the input port to the output port of the NonBlockingWrite class.

There is also a mapping from dataflow execution semantics to the abstract transWrite(Non-Blocking). In the concrete description in Section 3.2.1.4, the transWrite(Non-Blocking) will be initiated whenever there is a token to transfer on the data input in of the NonBlockingWrite class.

3.2.1.4 Behavior/ Interface Association

The following relations exist between the description in vc/cmult.h and the interface definitions of Section 3.2.1.3.

Interface/ Port/ Action	C++
port_I_rt	Constructor parameter _in from the BlockingRW class
port_I (mapping to 1.0)	Constructor parameter _out from the BlockingRW class
port_I_req	ReqPort() from the abstractproto class on top of BlockingRW
port_I_ack	AckPort() from the abstractproto class on top of BlockingRW
port_P_rt	Constructor parameter _in from the NonBlockingRead class
port_P (mapping to 1.0)	Constructor parameter _out from the NonBlockingRead class
port_P_ack	AckPort() from the abstractproto class on top of NonBlockingRead
port_O_rt	Constructor parameter _in from the NonBlockingWrite class
port_O (mapping to 1.0)	Constructor parameter _out from the NonBlockingWrite class
messSense(1)	Test on Q.getSize() followed by Q.get(). Event is true when a) a token is present and b) the value of the token is 1.
messEmit(1)	Put a token of value 1 (Q.put())
transRead	Read a token from Q (Q.get())
transWrite	Put a token on Q (Q.put())

3.2.2 Layer 0.1

3.2.2.1 Data Types

3.2.2.2 Internal VC Behavioral Description

Layer 0.1 of the CMULT VC is a wrapping of the behavior of the 1.0 layer with RT level interfaces. The functional behavior is identical. In contrast to layer 0.2, this layer uses RT level simulation semantics for the different ports at RT-level. Layer 0.1 also introduces the timing view of the CMULT VC. This layer models the internal latency of the CMULT VC.

Layer 0.1 is implemented in the simulation environment as a 'peel' around layer 1.0. Only the interface refinements are designed, while the internals stay identical. Internal operation is explained in Section 3.2.0.2 and listed in vc/cmult.h. The interface refinements are explained in Section 3.2.2.3.1 and listed in Section 3.2.2.4.

~~3.2.2.3~~ Interfaces

3.2.2.3.1 Overview and General Properties

Figure C4 illustrates the interfaces present at Layer 0.1. The figure is almost identical to the one of Layer 0.2 (Figure C2); but, in addition, a clock port is present. This port brings in a global synchronization signal (the clock) that is needed to implement the data I/O and parameter interfaces in a clock-cycle true fashion. The clock port is part of each of the interfaces and does not represent an interface by itself. The VC contains the following interfaces:

Name	Description
`Inp_clk`	Reads input values to multiply
`out_clk`	Produces resulting multiplicant
`Parmrt_clk`	Reads parameter values for multiplication

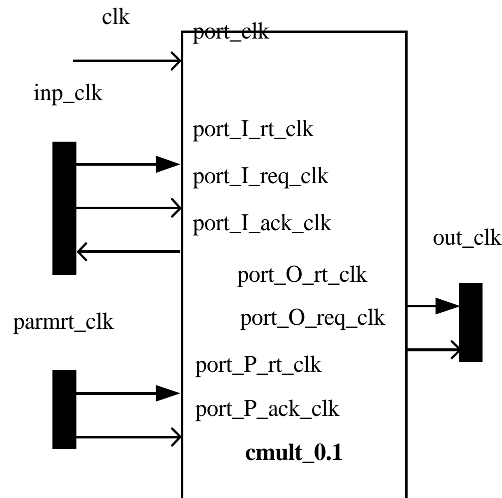


Figure C4: Layer 0.1 External View

- All datums of data nature flowing through `inp_clk`, `parmrt_clk`, or `out_clk` are `dfix_double`.
- Datums of control nature (flowing through `_req` and `_ack` ports) are of type `dfix_bool`.
- Datums flowing through `clk` are abstract; that is, they are pure events in the simulation environment itself.

The following transactions exist:

- `inp_clk` implements a blocking read, with `port_I_rt_clk` as the data signal, and `port_I_req_clk` and `port_I_ack_clk` doing the signalling. The transaction is equivalent with interface `inp` of layer 0.2.
- `parmrt_clk` implements a nonblocking read on `port_P_rt`, equivalent to the nonblocking read transaction of interface `parmrt` of layer 0.2.
- `out_clk` implements a nonblocking write, equivalent to the nonblocking write transaction of interface `out` of layer 0.2.

Layer 0.1 encapsulates 1.0 by extending the functionality of the interfaces and keeping the functional core identical. Each of the interfaces inp_clk, parmrt_clk and out_clk have a similar counterpart in Layer 0.2, with one important difference.: The interface blocks of layer 0.1 have an explicit notion of time, which is controlled through a control interface signal CLK. As shown in Figure C5, the complete VC is hybrid, since the functional core (taken from layer 1.0) does not has this control signal. As time cannot be transmitted through this Layer 1.0, an internal control signal *accept_token* and a delay block delay_1 is used. The delay inside of this block corresponds to the latency of layer 0.1 to evaluate internal behavior. Thanks to the *accept_token* signal and delay_1, the input and output interfaces are synchronized in *time*.

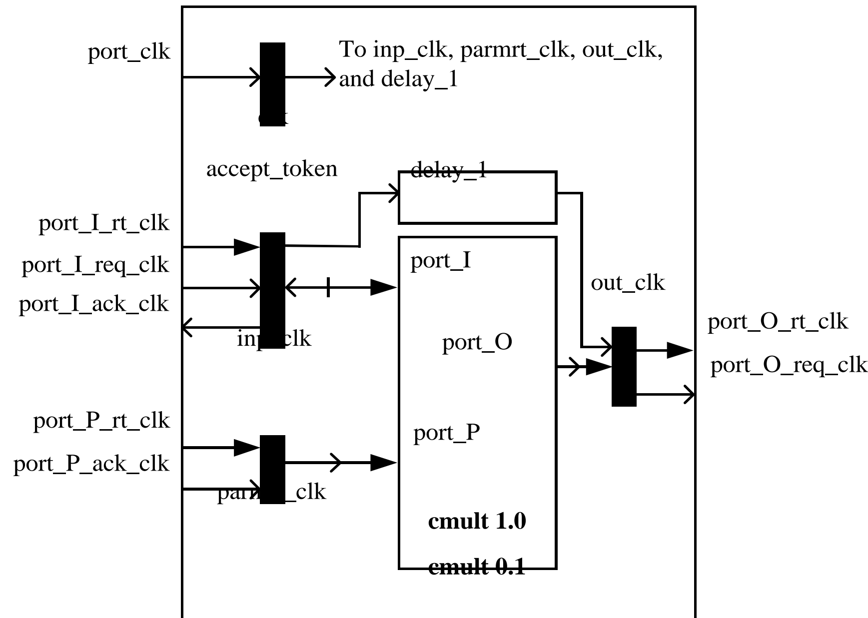


Figure C5: Layer 0.1 Internal View

In vc/df2rt_clock.h, C++ code for each of the interfaces is shown. Generally speaking, this code represents a clock-cycle true model with the following features:

- The overall processing is modeled by an FSMD (FSM + datapath). Control processing is expressed in a finite state machine, which models control states and transitions. Data processing is expressed in datapath instructions, which models set of expressions on signals. During FSMD instructions, the model indicates which signals are executed. The FSMD model is a one-clock-cycle-per-transition model. For each clock cycle, one or more datapath instructions can be executed.
- The following C++ classes are used by this FSMD model

`sig`	datapath signals (regs, constants, wires)
`sfg`	datapath instructions
`state`	control states
`ctlfsm`	finite state machine
`cnd`	state transition condition

- The use of these classes instructs the simulation environment on the structure of the FSMD model. When the simulation runs, the code is not directly executed. Rather, the data structure created by using the classes is executed. This meta-aspect explains why the FSMD code actually appears in the constructor of an object, rather then in a run() method.

The code in vc/df2rt_clock.h describes how the interface blocks of Figure C5 are actually designed. The correspondence of this figure and the class definitions are as follows:

Class	Description
Rtshell	This block models the latency of the Layer 0.1 functional core. Internally, it models this by creating a 1-bit wide chain of registers. For each clock cycle, an event should be accepted, e.g., a '1' is shifted into this chain. Otherwise, a '0' is shifted into it. An event is accepted by rtshell by executing the `triggers()` method in the driving FSMD (the interface inp_clk in our case). When the event is absent during a particular clock cycle, `triggersnot()` should be executed instead. Both of these methods actually translate to a datapath instruction created by the rtshell class. In the accepting FSMD (the interface out_clk in our case), the method `present()` is used to check for the presence of the event. The datapath instructions returned by `triggers()` and `triggersnot()` are created in the constructor of rtshell: seen as the few lines of code after `trigsfg.starts()`and `otrigsfg.starts()`.
abstractproto_cl ock	This class is functionally similar to the abstractproto class of Section 3.2.1.4, i.e., it transfers data from an input to an output port (FB in and FB out) under control of signalling done at two handshake ports (FB ackPort and FB reqPort). The class collects all datapath instructions that are necessary to implement the protocols of the interfaces (inp_clk, parmrt_clk, and out_clk). – read_ack reads the value of the acknowledge port and stores this into an acknowledge register (ack_reg). – set_req0 generates a '0' at the request port reqPort. – set_req1 generates a '1' at the request port reqPort. – reset_ack clears the value of ack_reg. – cpy_data performs the actual transfer from the in port to the out port.

blockingRW_clock	This class implements a blocking protocol. It actually models only the controller of this protocol, while it relies on the datapath instructions created in abstractproto_clock. Besides the implementation of the protocol, this class also creates events for the rtshell. The controller is described in the method. Three state transitions are described (the two states of this controller are data members). The first state transition starts in state s0 (the initial state, as indicated in the BlockingRW_clock constructor), and unconditionally (always) jumps to state s1. While making this transition, three datapath instructions are executed: - set_req0: a request is generated. - read_ack: the value on the acknowledge port is read. - RT.triggersnot(): No data is transferred, so rtshell is given no event. The second state transition describes the case when data should be transferred. It starts at state s1 and loops to state s1. The condition tested is the presence of a handshake acknowledge in ack_reg. The datapath instructions executed are: - set_req1: no request is generated. - read_ack: the value on the acknowledge port is read. - cpy_data: the data is transferred from the inp port to the out port. - RT.triggers(): An event is raised to the rtshell. The third transition describes the case when no data should be transferred. It starts at state s1 and loops to state s1. - set_req0: no request is generated. - read_ack: the value on the acknowledge port is read. - RT.triggersnot(): No event is raised to rtshell. This class will be used for the protocol running on interface inp_CLK.
NonBlockingRead_clock	This class is a stripped down version of the previous one, and corresponds functionally to the NonBlockingRead class of Section 3.2.1.4. The differences to the BlockingRW_clock class are: - no request signals are generated. - no rtshell events are generated. The class will be used for the protocols running on the interface parmrt_CLK.
NonBlockingWrite_clock	This class is a stripped down version of the BlockingRW_clock class, and corresponds functionally to the NonBlockingWrite class of Section 3.2.1.4. The class also accepts events from the rtshell block (which are used as transition conditions). The differences to the BlockingRW_clock class are: - no acknowledge signals are read. The class will be used for the protocols running on the interface out_CLK.

3.2.2.3.2 Interface-specific Description `inp_clk`

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
port_I_rt_clk	Cons	dfix_double	Reads Input I
port_I_req_clk	Prod	dfix_bool	Generates request signal
port_I_ack_clk	Cons	dfix_bool	Reads acknowledge signal
port_clk	Cons	dfix_bool	Introduces sequencing control event

STRUCTURAL – INTER-LAYER STATIC MAPPING

Layer 0.2 port	Layer 0.1 Mapping
port_I_rt	port_I_rt_clk
port_I_req	port_I_req_clk
port_I_ack	port_I_ack_clk
	port_clk

BEHAVIORAL – PORT ATTRIBUTES

See Figure C4.

BEHAVIORIAL – PORT TRANSACTIONS

Port	Transaction
port_I_rt_clk	transRead(initiator)
port_I_req_clk	transWrite(initiator)
port_I_ack_clk	transRead(initiator)
port_clk	messSense(anyEvent)

BEHAVIOR AL – INTER-LAYER BEHAVIORAL MAPPING

See Figure C5.

BEHAVIORAL – PROTOCOL DESCRIPTION

```
port_I: transRead(Blocking) {
   port_clk: messSense()
   port_I_req_clk: transWrite(1)
   C1 = port_I_ack_clk:transRead()
   DO {
     port_clk: messSense()
IF (C1) {
port_I_req_clk: transWrite(0)
          port_I_rt_clk: transRead
} ELSE
port_I_req_clk: transWrite(1)
C1 = port_I_ack_clk: transRead()
    } WHILE (NOT (C1))
}
```

Considering the code in vc/df2rt_clk.h, this protocol can be found in (part of) the state machine description of class BlockingRW_clock. (In addition to this protocol, this class also generates control events for the rtshell class.).

3.2.2.3.3 Interface-specific Description parmrt_clk

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
port_P_rt_clk	Cons	dfix_double	Reads Input P
port_P_ack_clk	Prod	dfix_bool	The presence of an acknowledge signal induces reading of P
port_clk	Cons	dfix_bool	Introduces sequencing control event

STRUCTURAL – INTER-LAYER STATIC MAPPING

Layer 0.2 port	Layer 0.1 Mapping
port_P_rt	port_P_rt_clk
port_I_ack	port_P_ack_clk port_clk

BEHAVIORAL – PORT ATTRIBUTES

See Figure C4.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
port_P_rt_clk	transRead(initiator)
port_P_ack_clk	transWrite(initiator)
port_CLK	messSense(anyEvent)

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

See Figure C5.

BEHAVIORIAL – PROTOCOL DESCRIPTION

```
port_P: transRead(Non-Blocking) {
  port_clk: messSense()
  C1 = port_P_ack_clk:transRead()
  port_clk: messSense()
  DO {
    port_P_rt_clk: transRead
    C1 = port_P_ack_clk: transRead()
} WHILE (NOT(C1))
}
```

Considering the code in vc/df2rt_clk.h, this protocol can be found in the state machine description of class `NonBlockingRead_clock.`

3.2.2.3.4 Interface-specific Description `out_clk`

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
port_O_rt_clk	Prod	dfix_double	Writes result
port_O_req_clk	Prod	dfix_bool	Generates request signal
port_clk	Cons	dfix_bool	Introduces sequencing control event

STRUCTURAL – INTER-LAYER STATIC MAPPING

Layer 0.2 port	Layer 0.1 Mapping
port_O_rt	port_O_rt_clk
port_O_req	port_O_req_clk port_clk

BEHAVIORAL – PORT ATTRIBUTES

See Figure C4.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
port_O_rt_clk	transWrite(initiator)
port_O_req_clk	transWrite(initiator)
port_clk	messSense(anyEvent)

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

See Figure C5.

BEHAVIORAL – PROTOCOL DESCRIPTION

```
port_O: transWrite(Non-Blocking) {
  port_clk: messSense()
  IF PROCEED
          port_O_req_clk: transWrite(1)
          port_O_rt_clk: transWrite
ELSE
          port_O_req_clk: transWrite(0)
}
```

This protocol corresponds to the finite state machine description of class `NonBlockingRead_clock` in vc/df2rt_clk.h. The PROCEED is a VC-internal flag that decides when the nonblocking write should proceed (i.e., the source of data that feeds port_O_rt_clk actually has data available). The PROCEED condition in layer 0.2 is dictated by the `rtshell` class output.

3.2.2.4 Behavior/Interface Association

The following relations exist between the description in vc/df2rt_clock.h and the interface definitions in Section 3.2.2.3:

Interface/ Port/ Action	C++
Delay in Figure C5	Class rtshell
transRead(initiator) transWrite(initiator)	These are created in the *datapath instructions* created by the abstractproto_clock constructor. These datapath instructions are used in turn by the FSMD descriptions in classes `BlockingRW_clock`, `NonBlockingRead_clock`, `NonBLockingWrite_clock` Note that the full FSMD description uses a hierarchy of objects: fsm, state, sfg (datapath instructions) and sig. This allows having datapath instructions defined in one class and used in another.
messSense(anyevent)	class `clk`. The C++ description is purely synchronous and therefore has only an abstract notion of a clk. The messSense is implicit since the RT-level simulation only runs when the clock ticks, i.e. when messSense(anyEvent) is true.
PROCEED	`RT.triggers()` or `RT.triggers-not()`. This condition is an output value of the RT class, which is of type `RT`.
`port_I_rt_clk`	Constructor parameter in of class `BlockingRW_ clock`
`port_I_req_clk` `port_I_ack_clk`	`set_req0, set_req1, ack_req` in `abstractproto_clock`
`port_P_rt_clk`	Parameter `_in` in `NonBlockingRead_clock`
`port_P_req_clk`	see `port_I_req_clk`
`port_O_rt_clk`	Parameter _out in `NonBlockingWrite_clock`
`port_O_req_clk`	see `port_I_req_clk`
`port_clk`	Is abstract in the C++ code. The type representing the clock signal is class `clk`

3.2.3 Layer 0.0

3.2.3.1 Data Types

Datum	`FX9`
Datavalue	-256 to 255
Datasize	9 bits

Datum	`Std_logic`
Datavalue	0 or 1
Datasize	1 bit

3.2.3.2 Internal VC Behavioral Description

Layer 0.0 is a VHDL view of the cmult block. The main purpose of this layer is to introduce explicit hardware semantics (reset behavior) into the VC description. The behavior is functionally equivalent to that of Layer 0.1. At reset, the parameter register will be set to zero. The VHDL view vc/cmult.vhd was generated out of a fully refined C++ view vc/cmultrt.h.

3.2.3.3 Interfaces

3.2.3.3.1 Overview and General Properties

Figure C6 presents an overview of the interfaces present at Layer 0.0. A further refinement with respect to the 0.1 Layer (Figure C4) is to introduce a reset interface and a clock interface. These two interfaces feed global control signals into the HDL view that embodies the cmult VC at Layer 0.0. Since reset and clock affect each of the other interfaces, as well as the internal behavior, they are provided through an interface of their own. The VC contains the following interfaces:

Name	Description
Inp_clk	Reads input values to multiply
Out_clk	Produces resulting multiplicant
Parmrt_clk	Reads parameter values for multiplication

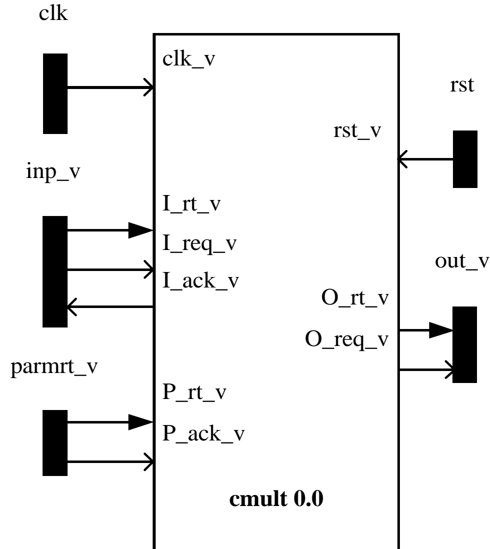


Figure C6: Layer 0.0 External View

- All datums of data nature flowing through inp_v, parmrt_v, or out_v are FX9.
- Datums of control nature (flowing through _req and _ack ports) are of type std_logic.
- Datums flowing through clk and rst are of type std_logic.

The following transactions exist:

- inp_v implements a blocking read, with I_rt_v as the data signal, and I_req_v and I_ack_v doing the signalling. The transaction is equivalent with interface inp of Llayer 0.2.
- parmrt_v implements a nonblocking read on P_rt_v, equivalent to the nonblocking read transaction of interface parmrt of Llayer 0.2.
- out_v implements a nonblocking write, equivalent to the nonblocking write transaction of interface out of Layer 0.2.
- clk and rst implement abstract transactions. An event at these inputs will advance time or reset the block respectively.

3.2.3.3.2 Interface-specific Description rst

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
rst_v	Cons	std_logic	Resets the block

STRUCTURAL – INTER-LAYER STATIC MAPPING

Not applicable.

BEHAVIORAL – PORT ATTRIBUTES

See Figure C6.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
rst_v	messReset

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

Not applicable.

BEHAVIORAL – PROTOCOL DESCRIPTION

Considering the code in vc/cmult.vhd, this protocol is present in the asynchronous reset of the block.

3.2.3.3.3 Interface-specific Description clk

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
clk_v	Cons	std_logic	Resets the block

STRUCTURAL – INTER-LAYER STATIC MAPPING

Not applicable.

BEHAVIORAL – PORT ATTRIBUTES

See Figure C6.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
clk_v	messSense(anyEvent)

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

Not applicable.

BEHAVIORAL – PROTOCOL DESCRIPTION

Considering the code in vc/cmult.vhd, this block represents the clock signal of the block.

3.2.3.3.4 Interface-specific Description inp_v

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
I_rt_v	Cons	FX9	Reads Input I
I_req_v	Prod	Std_logic	Generates request signal
I_ack_v	Cons	Std_logic	Reads acknowledge signal

STRUCTURAL – INTER-LAYER STATIC MAPPING

Layer 0.1 port	Layer 0.0 Mapping
port_I_rt_clk	I_rt_v
port_I_req_clk	I_req_v
port_I_ack_clk	I_ack_v

BEHAVIORAL – PORT ATTRIBUTES

See Figure C6.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
I_rt_v	messRead(initiator)
I_req_v	messWrite(initiator)
I_ack_v	messRead(initiator)

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

Not applicable

BEHAVIORAL – PROTOCOL DESCRIPTION

Not applicable

3.2.3.3.5 Interface-specific Description `parmrt_v`

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
P_rt_v	Cons	FX9	Reads Input P
P_ack_v	Prod	std_logic	The presence of an acknowledge signal induces reading of P

STRUCTURAL – INTER-LAYER STATIC MAPPING

Layer 0.2 port	Layer 0.1 Mapping
port_P_rt	P_rt_v
port_I_ack	P_ack_v

BEHAVIORAL – PORT ATTRIBUTES

See Figure C6.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
P_rt_v	messRead(initiator)
P_ack_v	messWrite(initiator)

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

Not applicable

BEHAVIORAL – PROTOCOL DESCRIPTION

Not applicable

3.2.3.3.6 Interface-specific Description `out_v`

STRUCTURAL – PORT IDENTIFICATION

Name	Role	Data Format	Description
O_rt_v	Prod	FX9	Writes result
O_req_v	Prod	Std_logic	Generates request signal

STRUCTURAL – INTER-LAYER STATIC MAPPING

Layer 0.2 port	Layer 0.1 Mapping
port_O_rt	O_rt_v
port_O_req	O_req_v

BEHAVIORAL – PORT ATTRIBUTES

See Figure C6.

BEHAVIORAL – PORT TRANSACTIONS

Port	Transaction
O_rt_v	messWrite(initiator)
O_req_v	messWrite(initiator)

BEHAVIORAL – INTER-LAYER BEHAVIORAL MAPPING

Not applicable

BEHAVIORAL – PROTOCOL DESCRIPTION

Not applicable