Formal Specification of the Virtual Component Interface Standard in the Unified Modeling Language

Annette Bunker and Ganesh Gopalakrishnan

UUCS-01-007

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

June 14, 2001

Abstract

As part of our charge from the Virtual Sockets Interface Alliance we search for a notation in which standards documents can be precisely specified. We approach the specification for standard problem in the context of the Virtual Component Interface Standard. We propose six orthogonal axes of specification as guides to creating a cohesive, well-rounded requirements specification. We then specify the Virtual Component Interface Standard in the Unified Modeling Language and evaluate that specification based on our six axes.

1 Introduction

The recent excitement over system-on-chip (SoC) systems raises the bar for formal methods techniques and tools. Not only do we face the usual challenges of keeping development costs low, keeping time-to-market small and ensuring high quality hardware, but now, we also face the challenge of integrating externally developed hardware and accompanying proofs into our overall design and verification strategy.

[1]This work was supported by National Science Foundation Grants CCR-9987516 and CCR-0081406

The Virtual Sockets Interface Alliance (VSIA), a consortium of SoC-interested companies, proposes to alleviate those concerns through development and adoption of appropriate SoC-related standards. As part of their ongoing work, the VSIA suggested that we adopt or create a notation suitable for formal specification of their standards and develop accompanying technology to enable compliance verification. As a first step toward this goal, we completed a series of case studies directed at identifying formal specification and formal verification issues related to one VSIA standard, the Virtual Components Interface (VCI) [Gro00].

Though the use of formal specifications for standards is generally ignored, standards are a good target for formalization, for several reasons. First, informal specifications are ambiguous. One vendor may understand the standard in one way, while another implements the another alternative, destroying the interoperability the standard was intended to create. Second, since standards are the fountainheads for many hardware realizations, errors in the specification are costly. Formal specifications admit to consistency checking and other debugging techniques, before the errors can propagate to designs. Finally, by specifying the standard formally, we open the door to formal standard compliance verification.

This paper reports on a case study in specifying the VCI, using the Unified Modeling Language (UML) [BRJ99]. We present our key contributions in depth in Section 5, but summarize them here for reader convenience.

- **six specification axes**: We propose six orthogonal axes of specification representing a minimal classification of points that must be considered in a standard specification such as the VCI.
- **UML specification of the VCI**: We present and explain the specification of the VCI standard in the Unified Modeling Language.

Furthermore, we consider social hurdles slowing the wide acceptance of formal specification. We then evaluate the UML as a potential formal specification language in terms of the six specification axes and its ability to overcome social problems.

Section 2 discusses other work being done in this area. Section 3 introduces the Virtual Component Interface Standard. Section 4 briefly describes our previous VCI-to-PCI bus wrapper verification case study and the lessons learned from it. We define the six specification axes and demonstrate our use of the UML as a hardware specification language for the VCI in Section 5. Finally, analysis and plans for future work make up Sections 6 and 7, respectively.

2 Related Work

While a large body of previous work exists in the context of using the UML as a formal specification language, to our knowledge, none considers its suitability as a specification language for hardware, as ours does. Below, we summarize the work that has been done in the area of using UML as a formal software specification language.

In [Kri00], Krishnan demonstrates one approach to using the UML as the basis for a formal specification by translating the diagrammatic notations to PVS state predicates. Once the diagram is embedded in the theorem prover and a notion of a trace is defined, a consistency check is equivalent to proving that there is a trace for the system which satisfies all the axioms generated from the UML diagram. The major advantage that this method promises is the ease with which it handles partial specification. Because Krishnan uses a past state temporal logic, it is often the case that previous events can be deduced from their later effects, even though they have not been explicitly specified.

Other methods similar to Krishnan's have been reported in the literature [AK99, AK98], though they use next state temporal logics and require complete information be specified in message sequence diagrams. We expect that compliance verification strategies built on this paradigm will be difficult to employ because of their basis in theorem proving and, as a result, will not gain wide acceptance.

Sendall and Strohmeier suggest translating use cases into operation schemas in [SS00b]. Operation schemas describe the effects of a system declaratively, using pre- and post-conditions. They also suggest some extensions to the Object Constraint Language, the first-order assertion language that is a part of the UML standard, to make it easier for designers to read and understand pre- and post-conditions, and presumably schemas, written in OCL in [SS00a]. Their proposed extensions do not enhance the expressibility of the language. They merely make the language look more like an imperative programming language and less like a declarative constraint language. While [SS00a] is a step in the right direction, we believe the authors do not take aggressive enough action. They only attempt to move one of the nine available UML diagrams into the realm of formal system specification.

Lilius and Paltor [LP99] contribute a formalization of the UML state machines that is the first to include all features available in the statechart diagram notation. Their associated tool, vUML, translates a UML statechart into the corresponding PROMELA and invokes SPIN on the model. If the model checker generates an error trace, it is translated into a sequence diagram for user consumption. While the advantage of their method is that no knowledge of SPIN or PROMELA is necessary to use the model checker via vUML, the

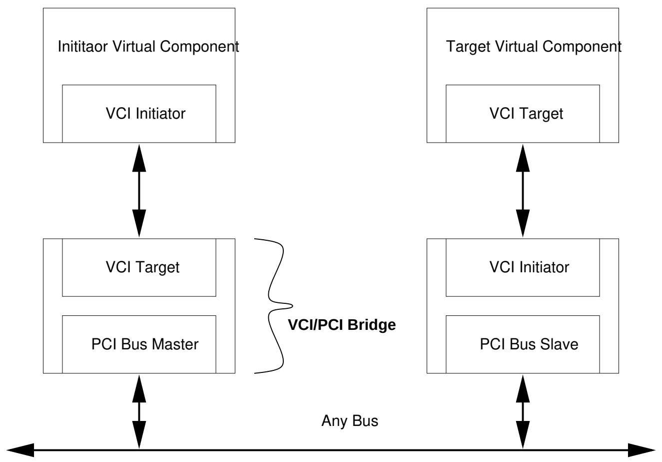


Figure 1: Two Virtual Components Connected by VCI Bus Wrappers and a Bus

major short coming of this work is that only statechart diagrams are currently formalized.

Finally, Övergaard and Palmkvist [ÖP98] describe an operational semantics for UML Use Cases in an object-oriented specification language named Odal. Again, their work is a step in the right direction, but it does not provide a formal verification engineer with enough tools for use in the hardware design arena.

3 VCI Overview

The Virtual Sockets Interface Alliance recently released the Virtual Component Interface standard. The VCI specifies an interface, rather than a bus and is intended to be used to connect two VCI-compliant virtual components together directly or over a bus, via VCI-compliant wrappers, as shown in Figure 1. This model allows the integrator to choose any bus, including proprietary busses, to connect virtual components that may have been designed by external organizations, by creating only the necessary bus wrapper modules.

The VCI standard describes a family of protocols. The Peripheral VCI (PVCI) is a subset of the Basic VCI (BVCI) which is a subset of the Advanced VCI (AVCI). The PVCI is intended for use with simple peripheral busses. It uses a two-signal handshake and is not a split-transaction protocol. The BVCI is a split transaction protocol which only designates that responses must return to the initiator in the same order in which matching requests were generated. The split-transaction AVCI tags requests and their responses so that multiple streams can be interleaved and transactions can be reordered.

Since we focus attention on the PVCI in this report of the UML case study, we describe it in more depth. PVCI transactions consist of cells and packets. A cell corresponds to a single handshake across the interface. A packet combines cells, like a bus burst, and must be transacted across the bus in a single arbitration cycle.

As the PVCI is not a split-transaction protocol, the request and response that transact a single cell are transferred simultaneously, on a single handshake. When the VCI initiator has a request, it raises VAL (valid) to signal this to the VCI target. When the target is able to respond to the request, it drives the response and raises ACK (acknowledge). All data is transferred while both signals are asserted. The initiator is required to keep request data steady while VAL is high and it is not allowed to back out a request once it has signaled one, except in time-out cases. Likewise, the target is required to keep response data steady while ACK is asserted. VAL and ACK must both be deasserted in the next clock cycle, unless another another transaction is being driven on the interface.

A PVCI request consists of address, byte enables, command (read or write), write data and end of packet marker (EOP). The end of packet field is set only if the current cell is the last cell in the current packet. Otherwise, the next cell address will be the logical successor of the current cell address. In the case of a read request, the write data field is don't-care.

A response consists of read data and response error fields. In the case of a write request, the read data field is don't-care. Response error denotes control errors on the interface.

4 VCI-to-PCI Wrapper Case Study

Previously, we formally verified a VCI-to-PCI bus wrapper, similar to the one shown in Figure 1. The purpose of this earlier case study was to learn the VCI standard thoroughly and to become familiar with the issues involved in carrying out a verification effort relating to a design derived from the VCI standard. Our wrapper was designed in Verilog and the verification was carried out using FormalCheck[2]. The VCI-complaint half of the wrapper met the BVCI specification. Detailed verification results are reported elsewhere [BG01][3]. For the purposes of the current report, however, we focus on the lessons learned from this case study.

[2]FormalCheck is a trademark of Lucent Technologies and Cadence Design Systems and is licensed to universities free of charge upon request: ftp.cadence.com/formalcheck.

[3]Complete design and Verilog code for the bus wrapper is available at http://www.cs.utah.edu/~abunker/vci/vlog.

1. English is a poor specification language because of its ambiguity and because items that are conceptually related are often found in remote portions of the specification.
2. A catalog-of-properties specification style also lacks conceptual cohesion. Furthermore, requirements written in this manner are often difficult to understand and implement.
3. Classes of requirements exist.
4. Grouping those requirements into classes may help alleviate completeness concerns as well as concept scattering.

Lessons 3 and 4, above, led us to identify six general classes of requirements from the VCI standard specification. We use these six classes, or specification axes, to guide our choice of specification notation, as well as our final specification. These axes are discussed in detail Section 5.1.

5 UML as a Hardware Specification Language

Outside the lessons learned about the technical aspects of the verification case study, we considered social aspects of specification and verification, as well. While the expressiveness of higher order logic is a boon to specifiers, in our experience, designers prefer not to read or create an implementation from it. Since the VCI specification is to become a standard from which many operational implementations are expected to spring, a widely readable specification is a must.

The readability requirement led us to consider graphical notations. We chose the Unified Modeling Language for its multiple viewpoint paradigm, graphical notation, widespread acceptance in the software community and growing presence in the hardware arena. The purpose of the specification case study is to evaluate the UML as a hardware specification language candidate.

The rest of this section describes, first, the six proposed specification axes and, second, the UML specification of the Virtual Components Interface Standard.

5.1 Six Axes of Specification

In the spirit of lessons 3 and 4 from the verification case study, we identified six general areas of interest to the design and verification teams as they approach a standard implementation project. These six areas are meant to serve as a guide to a specifier attempting to enumerate all requirements contained in the standard. We do not, however, claim that they guarantee specification completeness or consistency. We outline the six axes below, including a definition of each, as well as a few examples of items that fit into each category from the Peripheral VCI standard.

1. **Signaling and Timing**: This portion of the specification describes signal timing and meaning. This information is generally taken from timing diagrams and tables which define signal semantics in the informal specification. Examples of VCI-related issues that reside in this category include the VAL-ACK handshake behavior and timing and EOP=0 means that the next request will access the cell logically following the current cell.

2. **Data Handling**: This portion of the specification includes all aspects of the data path. Examples from the VCI standard include valid combinations of the byte enables and the property that cell addresses must remain within the address space of a single target for the entire packet.

3. **Transaction Characteristics**: This portion of the specification describes more general characteristics of the protocol such as split transaction characteristics, bus locking mechanisms, packetization, etc. In the PVCI, transaction characteristics include the cell-packet relationship and the coupling of request and response onto a signal control handshake.

4. **Global Properties**: Specification items from this category describe high-level invariants and algorithmic characteristics of the protocol. In the VCI specification, this category generally covers characteristics that are determined at component instantiation time, such the handling of VCs that support address widths that are not multiples of eight bits.

5. **Internal State**: This portion of the specification describes state maintained by each participating agent as required by the standard. There is no example of usage of this category from the VCI standard, however, other widely-used protocols, such as the Intel Pentium III (R) bus do require that agents complying with the protocol maintain certain bits of state information.

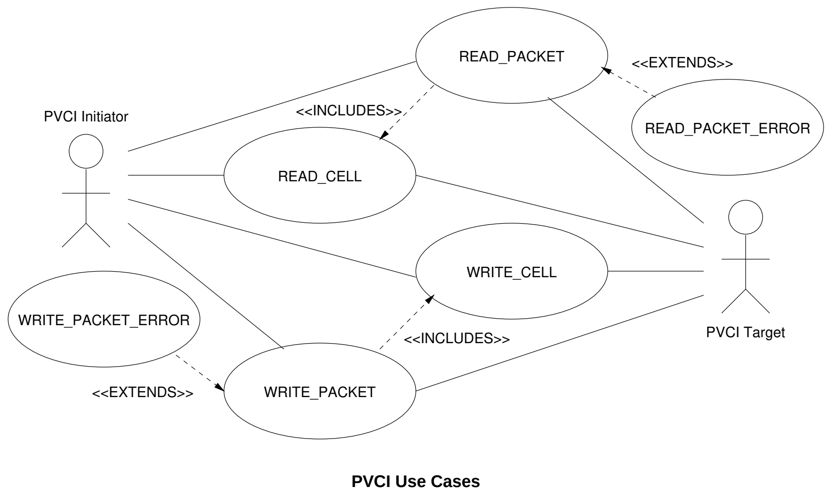


Figure 2: User View of the Peripheral Virtual Component Interface

6. **Error Handling**: This portion of the specification describes agent behavior in the face of erroneous input. Data errors may be described in either this section or in the data handling section.

5.2 Specifying the VCI in the UML

We used three types of UML diagrams to specify the PVCI. This subsection presents representatives of each type of diagram and discusses its features and specification power. Due to space limitations, we present elements of the PVCI specification. An analogous discussion of the BVCI specification would require detail and explanation that is beyond the scope of the current report, but is similar to the PVCI specification presented.

5.2.1 Use Case Diagrams

Figure 2 shows the use case diagram for the PVCI. Booch, et al define a use case as "a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor." [BRJ99] An actor is some user of the system, whether that user be human or automated. The purpose of a use case diagram, then, is to enumerate the scenarios in which the system might be used and which actor(s) will interact with each scenario. The use case diagram for the Peripheral Virtual Component Interface shows two actors, representing the initiator virtual component and the target virtual component, respectively. Even though actors may be mechanical in nature, they are represented as stick figures in the UML and labeled, as shown. We connect actors that may interact directly with a use case by an association, which is represented as a solid line.

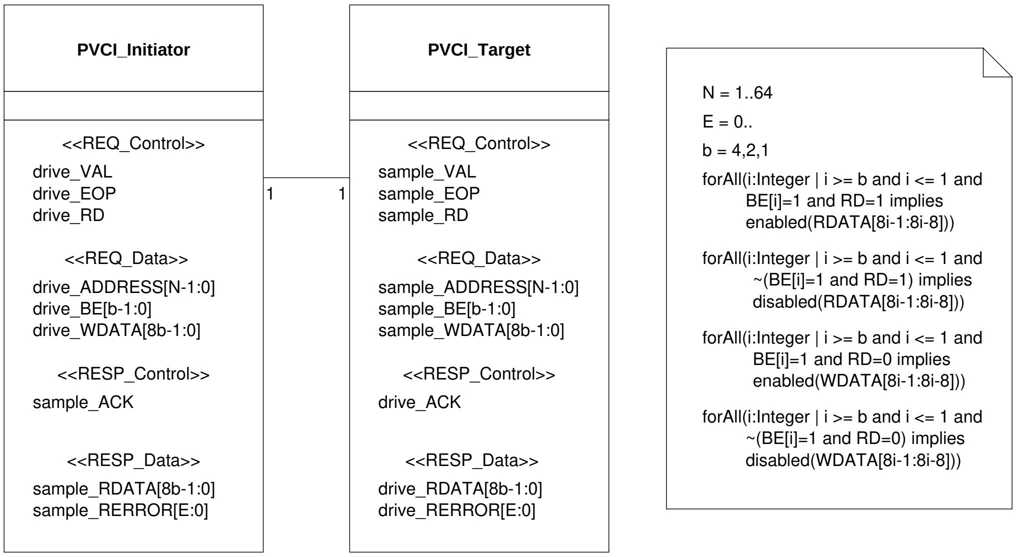
The PVCI only supports reads and writes, as shown in the diagram. The basic read or write transaction transfers one cell of data between initiator and target. Because reading or writing a packet of data that contains more than one cell across the PVCI essentially does several `READ_CELL` or `WRITE_CELL` operations, we use the UML standard stereotype "<<INCLUDES>>" to show this. Because the VCI standard only specifies that errors be reported on a packet level, we show the error scenario as an extension of `READ_PACKET` and `WRITE_PACKET` use cases by using the UML standard stereotype "<<EXTENDS>>." This variant use case may be read as, "The use case `WRITE_PACKET_ERROR` extends the functionality of the use case `WRITE_PACKET`."

5.2.2 Class Diagrams

Figure 3 shows the structural view of the PVCI standard. Each rectangle on the diagram represents a class. The first compartment within the class contains the class name, for instance `PVCI_Initiator`. The second contains object attributes. For protocol specification, we use this compartment only for state that the protocol changes. As the VCI protocol does not require any state of the objects involved, this is blank in this particular specification. We anticipate, however, that other protocols may make extensive use of this compartment.

The third compartment contains a description of each method a class contains. User-defined stereotypes, such as "<<REQ_Control>>," are used to organize the methods by function for readability and understandability. In this diagram, we use the notion of "drive" and "sample" to indicate the directionality of the ports represented by these methods. Methods beginning with "drive" are outputs while those beginning with "sample" are inputs. The portion of the method name in uppercase letters corresponds directly to the name of the signal given in the VCI standard specification.

Between the two classes, we represent an association because the two classes communicate with one another. The semantics of this association are exactly the same as those assigned to the association in the use case diagram, above. Notice, however, that this association has been adorned with multiplicities at each end. These multiplicities, both 1 in our example, indicate that for each VCI-compliant interface, there must be exactly one initiator and one target.



PVCI Class Diagram

Figure 3: Structural View of the Peripheral Virtual Component Interface

The dog-eared rectangle on the right of the diagram represents a note. Notes are allowed anywhere in UML diagrams. Usually, notes are attached to the element of the diagram that they address by a dashed line. We do not do so in this particular diagram, however, as the noted information applies to both classes in the diagram and the methods and variables in them.

The first item on the note states that N may range from 1 to 64, inclusive. When coupled with the information in the class itself, this tells us that a VCI address may be from 1 to 64 bits wide. Similarly, E ranges from 0 to infinity and b may take on the value 4, 2 or 1.

The remaining four items on the note specify invariant conditions on the relationship between the value of the RD (read) signal, the byte enables the read data bus and the write data bus. They are written in the first order language, Object Constraint Language (OCL), whose semantics are as one would expect. For example, the first constraint states that if a byte enable bit is set and the transaction is a read, then the corresponding byte lane of RDATA is enabled for read.

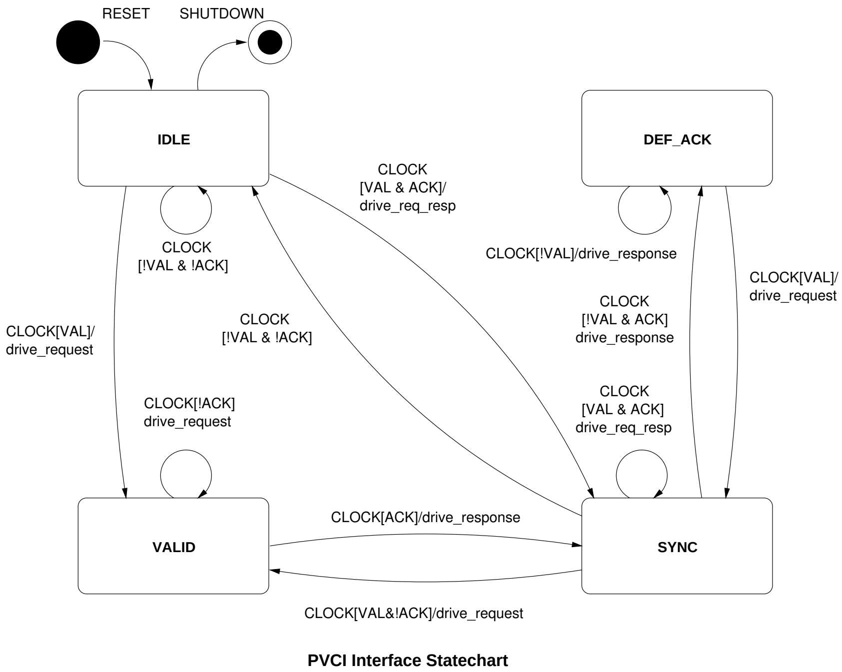


Figure 4: Clock-based Behavioral View of the Peripheral Virtual Component Interface

5.2.3 Statechart Diagrams

Finally, we choose to specify the behavioral aspects of the VCI from two different perspectives. The first looks at the interactions that happen across the interface itself. It describes what a third-party viewer would see by probing the interconnect between the initiator and the target. While it is event-driven in nature, all events are clock ticks. The second perspective, however, looks at the initiator and the target separately and describes the internal behavior of each agent. Time granularity is much smaller than a single clock-tick, allowing us to specify timing requirements precisely.

Figure 4 shows the first behavioral view of the VCI. The interface can be in one of four states, denoted by the rectangles with rounded corners. Transitions are triggered by the CLOCK event. The transition guards are noted inside square brackets ([]) and the action performed along the transition follows the forward slash (/). Note that time granularity is the clock tick in this diagram, so that precise ordering of request stability and valid need not be shown. It is sufficient at this level to show that data is being driven while the control line is asserted.

The pseudo-states at the top of the diagram represent the initial and final states and the transitions that lead to and from them. The solid, black circle is the initial state of the

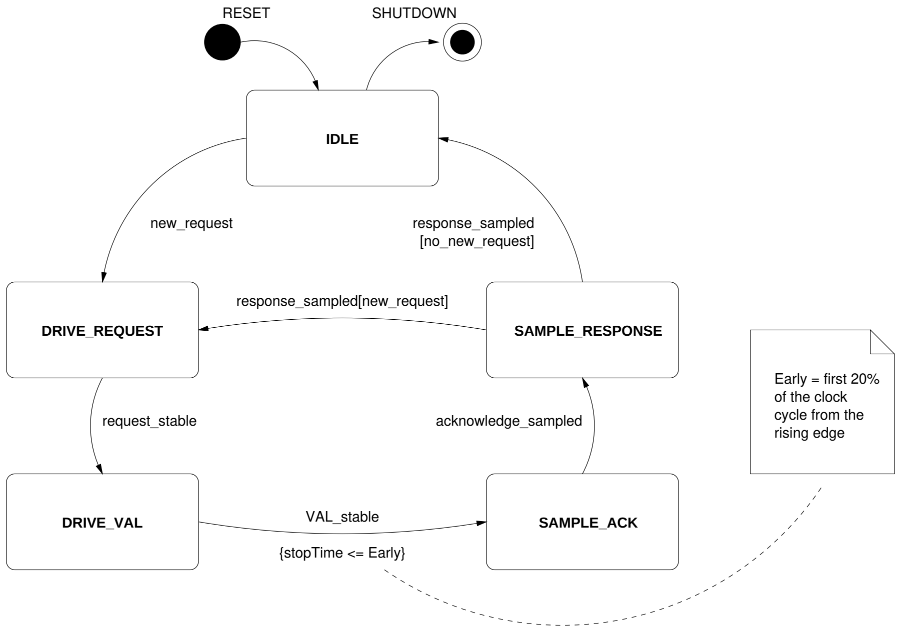


Figure 5: Behavioral View of the Peripheral Virtual Component Interface

diagram and the only transition that leads out of it is the `RESET` transition. The white- and black-banded circle represents the final state to which the only way we can get is by taking a `SHUTDOWN` transition.

Studying the English VCI specification and creating this diagram helped us see an issue in the published standard. In its original form, the standard allowed a transition from `IDLE` to the `DEF_ACK` state. While this transition is allowable in the analogous diagram of the BVCI because of the split transaction nature of the protocol, it is not allowable in the PVCI. The existence of this transition implies that the target can service a request before it is even made. In the BVCI, this transition means only that the target can accept the request for processing before it is made.

Figure 5 details the internals of the PVCI initiator. A diagram similar to Figure 5 exists for the PVCI target. The main difference between this diagram and the preceding one is the granularity at which events occur. This temporal spacing may be finer than that of the clock and events need not occur at regular time intervals. We choose to create a second model at this level of detail to specify the fine details about event ordering that we could not express at the clock level. For instance, this diagram requires that the request become stable on the interface lines before VAL becomes stable. Similarly, the initiator is not allowed to sample the response before it samples ACK high.

Note the OCL constraint on the `VAL_stable` transition. The constraint is placed inside curly braces ({}) as required by the UML. This particular constraint uses the built-in OCL feature "stopTime" to require that VAL must be stable by Early. Again, we use a note to elaborate on an item that is not obvious from the picture. This time, we attach the note directly to the stopTime constraint. The note defines Early in this specification.

6 Analysis

To date, the VCI specification in the UML consists of fifteen diagrams. Five diagrams, roughly five pages, relate to the PVCI as compared to fifteen pages of English text, tables and timing diagrams. The remaining diagrams replace and extend the PVCI specification to create a BVCI specification. In all, eleven diagrams currently make up the BVCI standard specification, compared to thirty-three pages of informal specification.

The UML specification of the PVCI and BVCI took 6 person-weeks, including time to learn the UML and OCL. We expect to specify the AVCI in 1-2 person-weeks.

We use three of nine available types of diagrams, use case diagrams, class diagrams and statechart diagrams, to describe various aspects of the standard. Use case diagrams illustrate the user interface to the protocol. Class diagrams illustrate the structural view of the protocol agents and contain invariants over data structures. Statechart diagrams show the behavioral view of the system, along with timing constraints and behavioral invariants. Statecharts are useful for specifying behavior at differing levels of detail.

As stated earlier, we expect to find a specification technique which supports us in specifying the protocol along six axes. Below, we evaluate the use of the UML according to its practicality along each proposed axis.

1. **Signaling and Timing**: Statechart notation completely specifies all signaling and timing requirements. Though some believe that timing diagrams are necessary to complete this aspect of specification and should be added to the UML [Goe00], we disagree. OCL adornments can precisely specify timing requirements such as output deadlines.

2. **Data Handling**: The combination of class diagrams and state chart diagrams can fully express all aspects of data handling. Data/data and data/control dependencies can be demonstrated in the static view of the system, while data-related timing issues can be shown with the dynamic view of the system.

3. **Transaction Characteristics**: Because the PVCI is not a split-transaction protocol, these sorts of issues were minimal in its specification. Issues of this sort remain to be addressed in the BVCI specification, but we believe that we can do so by using message sequence charts, accompanied by OCL notations, and model the protocol at a slightly higher abstraction level than that seen in this discussion.

4. **Global Properties**: Of the six axes, the handling of global properties is the least palatable in our UML specification. Global properties are currently scattered throughout the specification and specified in many different formats. They are sometimes implicitly described in behavioral diagrams, sometimes explicitly specified (as in the case of signal stability deadlines) and in some cases, completely separate OCL description is necessary.

5. **Internal State**: While the VCI specification case study does not support us, we believe that required internal state can adequately be represented as attributes in class diagrams.

6. **Error Handling**: The current UML specification does not include error handling at any level of the VCI. However, we believe errors will integrate into the current behavioral model easily.

Not only should the language chosen to represent a formal specification have expressiveness to represent issues in these six areas, it should also be easy to learn and read, it should be widely accepted and it should integrate seamlessly into the implementation specification and documentation. Finally, it should enable formal verification of standard compliance.

We prefer the graphical format of the UML to the English as a specification language not only for its conciseness, but because logically related requirements are physically close together in the specification and it is less ambiguous than English, even without a fully-defined formal semantics. We prefer the UML to higher order logic specifications and catalog-of-properties specifications for its ease of understanding and its growing acceptance and tool support. While work must be done in order to make the Unified Modeling Language a formal solution to all specification issues raised here, it shows strong promise for use as a formal hardware specification language.

7 Future Work

We intend to continue investigations into the potential use of the UML as a formal hardware specification language. We aim, particularly, to help the VSIA provide its members with formal solutions to standards compliance problems. Specifically, our goals for this continuing work include:

- Complete a UML specification of the AVCI.
- Add error-handling and transaction-level requirements to the UML specification of all levels of the VCI.
- Create a formal semantics for the UML.
- Create a formal consistency-checker for the UML documents.
- Develop a strategy for verifying compliance to protocols specified in the Unified Modeling Language.

References

[AK98] T. Aoki and T. Katayama. Unification and consistency verification of object-oriented analysis models. In *Asia Pacific Software Engineering Conference*, pages 296–303. IEEE Computer Society Press, 1998.

[AK99] T. Aoki and T. Katayama. How to support verification of object-oriented analysis models using HOL. In *Proceedings of Systems, Cybernetics and Informatics/Information Systems Analysis and Synthesis*, pages 525–532, 1999.

[BG01] Annette Bunker and Ganesh Gopalakrishnan. Verifying a virtual component interface-based PCI bus wrapper using FormalCheck. Technical report, University of Utah, June 2001.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley, 1999.

[Goe00] Richard Goering. UML language eyes system-level design role. www.eetimes.com, September 2000.

[Gro00] OCB Design Working Group. *VSI Alliance Virtual Component Interface Standard*. Virtual Sockets Interface Alliance, November 2000.

[Kri00] P. Krishnan. Consistency Checks for UML. In *Proceedings of the Asia Pacific Software Engineering Conference*, pages 162–169, December 2000.

[LP99] Johan Lilius and Ivan Porres Paltor. Formalising UML state machines for model checking. In Robert France and Bernhard Rumpe, editors, *UML'99 – The Unified Modeling Language: Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445. Springer Verlag, October 1999.

[ÖP98] Gunnar Övergaard and Karin Palmkvist. A formal approach to use cases and their relationships. In Jean Bèzin and Pierre-Alain Muller, editors, *UML'98: Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 406–418. Springer-Verlag, June 1998.

[SS00a] Shane Sendall and Alfred Strohmeier. Enhancing OCL for specifying pre- and post-conditions. In *UML 2.0 - The Future of the UML Object Constraint Language (OCL)*, York, UK, October 2000. Workshop on UML 2000 – The Unified Modeling Language: Advancing the Standard, Third International Conference.

[SS00b] Shane Sendall and Alfred Strohmeier. From use cases to system operation specifications. In *UML 2.0 - The Future of the UML Object Constraint Language (OCL)*, York, UK, October 2000. Workshop on UML 2000 – The Unified Modeling Language: Advancing the Standard, Third International Conference.