Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL

Peter J. Ashenden
Dept. Computer Science
University of Adelaide, SA 5005
Australia
petera@cs.adelaide.edu.au

Philip A. Wilsey
Dept. ECECS, PO Box 210030
University of Cincinnati
Cincinnati, OH 45221-0030, USA
phil.wilsey@uc.edu

Abstract

This paper reviews the requirements on a language for modeling behaviour and structure at the system level, and considers possible approaches to extending VHDL to meet these requirements. Modeling issues that obtain in a system-level design language are identified, including abstraction of data, concurrency, communication and timing, and design refinement. Some system-level design languages and notations are surveyed, and previous proposals to extend VHDL for system-level design are reviewed. Specific language design issues for extending VHDL are discussed, and some alternative solutions are presented.

1. Introduction

One of the main challenges facing designers of complex integrated systems is that of managing complexity. Many systems are now composed of complex digital and analog hardware, managed by embedded software. In some cases, they must communicate with other systems, forming part of a larger distributed system.

The key to managing complexity in such systems is the use of abstraction. Designers focus first on the abstract properties of a system in various domains, and devise a systems architecture that will satisfy the requirements placed on the system. They may devise multiple architectures, analyze them, and choose among them based on the analyses. The domains under consideration include behaviour, structure, performance, physical arrangement and packaging, power consumption, thermal, cost, etc. In each domain, abstraction is used to focus on the major aspects of the system, and minor detail is ignored. Judicious choice of abstractions makes architectural design and analysis tractable, and aids subsequent partitioning and refinement of the system design.

A number of specialized languages may be used in designing a system. Some, such as the Unified Modeling Language (UML) [17, 35], are primarily notations for expressing the relationships between the various components that make up a system. Others, including those discussed in this paper, focus on expressing the required behaviour and logical organization of a system and its components. Once behaviour has been captured using such languages, the system can be simulated to verify that it meets its requirements. Subsequently, it can be manually or automatically refined to a more detailed implementation in terms of hardware, software, or a combination of both.

Hardware description languages also focus on describing systems in the behavioural and structural domains. However, due to their origin as languages for hardware design, they do not include strong capabilities for abstracting over data and for describing complex interactions. For example, in Verilog [26, 39], data types are closely bound to their binary representation, and signalling between modules includes aspects of electrical implementation. VHDL [2, 25], on the other hand, allows more abstract expression of data, since its type system is similar to that of conventional programming languages. However, its signalling features are still closely bound to electrical implementation.

This paper reviews the requirements on a language for modeling behaviour and structure at the system level, and considers possible approaches to extending VHDL to meet these requirements. Section 2 reviews the modeling issues that obtain in a system-level design language. Section 3 briefly surveys system-level design languages and notations that are widely used, and Section 4 surveys previous proposals to extend VHDL for system-level design. Section 5 focuses on specific language design issues for ex-

---

tending VHDL, and discusses some alternative solutions. Section 6 summarizes and presents conclusions.

2. Modeling Issues for System-Level Design

At the system level, a system can be modeled as a collection of active objects that react to events, including communication of data between objects and stimuli from the enclosing environment. Abstraction is needed in a number of areas to make system-level behavioural modeling tractable:

· abstraction of data,

· abstraction of concurrency, and

· abstraction of communication and timing.

It is important to note that the issues of concurrency and communication have been widely discussed in the literature on concurrent programming languages. (See Ball *et al* [7] for a survey.) We draw on experience from that field in considering the design of system-level modeling languages.

Early in the design flow, the representation of data in a system should be at a high level of abstraction. For example, data may initially be modeled using uninterpreted tokens, allowing performance of a design to be estimated based on token flow rates and queuing delays. When data types are refined to concrete representations, abstract data types and object-oriented techniques [10] can be used to manage the increased complexity. Refinement to hardware-specific representations should be deferred to later in the design flow. Thus, a requirement for a system-level design language is a facility for expressing uninterpreted and abstract data types.

A system-level design language needs to allow expression of concurrent processes representing the active objects in a system. In some systems, the number of active objects is not statically determined, but may vary during operation of the system. For example, in a client/server system, new service agents may be created as requests arrive from clients, allowing multiple requests to be processed concurrently. In order to describe such systems, a system-level design language must allow expression of process types that may be dynamically instantiated and terminated.

At the system level, processes representing active objects must interact to communicate data and to synchronize their operation. The simplest form of interaction is message passing, involving the transfer of data from a sender process to a receiver. The act of message passing can also be used to synchronize the processes. We focus on message passing in this paper, as it a natural abstraction of communication common to both software and hardware. Other forms of interaction, such as rendezvous and remote procedure call are possible [7], but are oriented specifically toward software implementation. They can, however, be expressed in terms of message passing.

There are two ways in which message passing abstracts away from the details of communication in hardware description languages. First, communication events are not tied to specific times, but rather are simply ordered by relative time of sending. This causality-based ordering is weaker and less constraining than clock-time ordering, and so is more appropriate at the early stages of design. Second, communication events may be queued (either by queuing messages or processes), rather than relying on the recipient sensing data at the correct time. This allows multiple communication events to form a stream or a transaction without the need for detailed signalling protocols.

System-level design occurs at the early part of a process that involves refinement to lower-level hardware and software implementations [18]. A model at the system level should be expressed in a form that enables verification that a refinement correctly implements the model. Possible approaches include behavioural synthesis [32] (correct by construction), and formal verification using model checking and equivalence checking [23, 33]. Refinement to a software implementation is facilitated by a system-level modeling language that is closely related to programming languages. In principle, the hardware and software implementations could be expressed in the same language as the system-level model, thus avoiding semantic mismatches between different languages in the design flow.

Given a system partitioned at the architectural level, it is important to specify the interfaces between the modules. Incorrect implementation of interfaces is a significant source of defects in system designs. Correct implementation is aided by explicitly modeling the interfaces, including the data representations used and the communication protocols. Explicitly modeled interfaces can also be used to aid refinement of the system, by describing transformations between higher-level and lower-level representations and protocols.

3. Review of System-Level Design Languages

There are several languages for system-level specification and design that are widely used. A number of them, such as Z [15, 36] and LOTOS [9, 28], focus on specifying a system in a declarative style. Others adopt a process-oriented approach to describing the structure and behaviour of a system at a high level of abstraction. In this section, we review four languages in the latter category, showing how they do, or do not, address the requirements discussed in Section 2.

3.1 StateCharts

StateCharts [20] is a graphical notation for expressing the behaviour of a system. Its abstraction of concurrency and communication is based on a extended finite state machine (FSM) model. Extensions include hierarchical states, in which a superstate represents a subgraph of the FSM; the

machine is in the superstate when it is in any of the sub-states. StateCharts also allows concurrent composition of states; the machine is in both of the states. If the concurrently composed states are hierarchical, those states act as concurrent submachines. The global machine is in some substate of each of the concurrently composed states. Thus, each concurrently composed state can be viewed as a concurrent process in the model. In StateCharts, these processes are statically specified using a graphical editor. The effect of a StateChart on its environment is specified by *actions* associated with states and transitions. Actions include starting and stopping *activities*, such as generating outputs on signals connected to the environment.

Transitions in a StateCharts FSM occur are triggered by *events*, and may optionally be guarded. An event may be input from the environment, a transition in a concurrently composed submachine, or a time-out. Guard expressions may be be predicates over the current state of concurrently composed submachines. As Harel comments [20, page 264], the use of the state of one submachine to cause events or guard transitions in other submachines implies a form of broadcast communication between the processes representing submachines. There is no explicit timing involved in the StateCharts notation beyond the causal ordering of transitions fired by events, with actions being performed instantaneously.

While the StateCharts notation addresses the issues of abstraction of concurrency, communication and timing, albeit with a somewhat limited set of abstractions, it does not adequately address abstraction of data. Using events for communication limits StateCharts to uninterpreted modeling of data. Thus, the notation is mainly useful for modeling the control-oriented aspects of a system. The notation also does not deal with modeling of the interfaces between components. Indeed, it violates the principles of modularity and information hiding (see Booch [10]) by allowing state information nested inside of one submachine to be names in event and guard expressions in other submachines.

The StateCharts notation has been adopted in the Uniform Modeling Language (UML) [17, 35]. In that version of the notation, events can be parameterized, allowing communication of data between processes via events; states can have variables; and actions can be procedural, allowing update of variables and sending of messages to objects.

3.2   Estelle

Estelle [11, 29] is an ISO standard language for describing distributed information processing systems. The concurrency aspects of a system are described using a hierarchy of instances of *modules*, each of which is a concurrent process. A module has *interaction points* for communication, and may have locally declared variables

and nested module instances. The behaviour of a module is specified in terms of an extended non-deterministic state machine, with Pascal-like statements included as transition actions. Transition conditions are expressed in terms of message arrival at an interaction point, guarded by boolean expressions. Modules may be statically instantiated, or may be dynamically instantiated as part of transition actions. The modules within a group of processes called a *subsystem* execute with synchronous parallelism; first, enabled transitions are selected, then the selected transitions are executed. Modules in different subsystems are permitted to execute with asynchronous parallelism.

Communication in an Estelle description takes place using buffered asynchronous message passing over typed links between interaction points. The links may be statically created, or dynamically created as part of transition actions. Estelle provides for abstraction of interfaces in the form of *channels*, which specify a protocol for message exchange over linked interaction points. A channel defines a set of roles, and, for each role, a set of allowed messages that may be communicated. An interaction point of a module may be declared to take on a particular role of a declared channel type. While Estelle does provide richer facilities for expressing data than StateCharts, they do not extend to expression of abstract data types.

3.3   SDL

SDL [16, 31], like StateCharts, is a language for describing the behaviour and structure of systems. The language has both textual and graphical notations, with the same underlying semantics. A system is described as a statically specified hierarchy of *process sets*. Within each set, processes may be either statically or dynamically created instances of declared *process types*. Process sets are connected with statically specified typed channels (*signal routes*), over which the processes may communicate using buffered asynchronous message passing. A process may have locally declared variables. The behaviour of a process is expressed in terms of an extended finite state machine model. Transitions are enabled by message arrival, and possible actions on transitions include variable assignment, process creation, sending a message, and a form of remote procedure call.

SDL has sophisticated features for abstracting over data types, based on the ACT-ONE model for defining abstract data types (ADTs) using axiomatic semantics [13]. A number of types are predefined, including numeric, boolean, character and time types. The model also allows specification of composite types, such as record, array and list types.

The revision of the SDL standard in 1992 enhanced the language to include object-oriented features. For example, process types may be specialized through inheritance; the specialized process type inherits the variables, states and

transitions of the parent, and may add new variables, states or transitions. Data types and signal types may also be specialized through inheritance, with new elements being added to the specialized types.

3.4 CSP

Hoare initially proposed Communicating Sequential Processes (CSP) as a programming language for concurrent systems [21], and subsequently developed the notation into a formal mathematical theory of communicating concurrent systems [22]. In the programming language form, a system is described as a set of statically specified processes that communicate through statically specified channels using synchronous message passing. Each process contains sequentially executed actions, including variable assignment, message sending and message reception. While the language does provide abstraction of concurrency, communication and timing, it does not support abstraction of data beyond a few primitive data types, nor abstraction of interfaces. The original formulation of the language was not intended as a full-scale language for serious use. Instead, it formed the basis from which the mathematical theory was developed. While the original form of CSP lead to the programming language OCCAM [27], the mathematical theory is now used as a formal semantic theory for programming languages, and for formal specification of software systems.

4. Previous Proposals for Extending VHDL

In previous papers [3, 4] we have reviewed a number of proposals that suggest using object-oriented language features to support system-level modeling. They seek to represent a system as a set of objects that communicate by invoking operations in other objects.

Cabanis *et al* [12] add a class construct to the language, similar to the class construct in C++, and specify a weak form of concurrency control for managing access to shared objects by multiple processes. The Vista OO-VHDL language described by Swamy *et al* [37] and the OOVHDL language described by Benzakki and Djafri [8] both extend the notion of a VHDL entity, viewing it as a form of class that can include operations to be invoked by processes in other design units. The Objective VHDL language described more recently by Radetzki *et al* [34] also follows this approach, but does not specify the means of communication by which operations are remotely invoked. Instead, they assume that the designer will customize a communication protocol from a library to provide the required communication, and have the receiving object dispatch to the operation. In practice, their language design makes this quite cumbersome, detracting from abstraction otherwise afforded.

Other proposals extend VHDL with declarative constructs for specifying the behaviour and other aspects of a system. In VSPEC [1], for example, required behaviour is described using axiomatic specification techniques, and constraints are specified in the form of relations over constraint variables. Including these specifications in a model allows tools to automatically verify that an implementation meets the specifications. Another proposed extension, VHDL+ [24], provides a mechanism for abstracting the interfaces between partitions in a system-level design, and for specifying refinements of interfaces to lower levels of abstraction. These proposals address the wider context of system-level modeling. Since this paper focuses on behavioural and structural aspects, we defer consideration of these proposals to a later paper.

5. Language Design Issues for Extending VHDL

In this section, we discuss ways in which VHDL might be extended to address the requirements for system-level behavioural and structural modeling laid out in Section 2. An important consideration in this discussion is ensuring that the extensions integrate cleanly with existing language mechanisms, and avoiding replication of mechanisms. While we illustrate possible extensions with examples, no concrete language proposal is implied.

5.1 Data Abstraction

The type system of VHDL is closely related to that of the Ada programming language [30], and allows for a significant degree of data abstraction above the level of binary-encoded values. For system-level modeling, abstract data types provide a further level of abstraction. VHDL has relatively weak facilities for describing abstract data types, and a number of proposals have been put forward for strengthening the language with object-oriented features. We have previously surveyed these proposals [4], and have proposed our own extensions [6] based on the facilities of Ada.

5.2 Communication Abstraction

In Section 2 we identified message passing as a more abstract form of communication than communication through signal assignment. Signals in VHDL can be viewed as statically instantiated, named communication channels. However, the semantics of passing values via signals is based on a low-level model of electrical implementation, and is significantly different from the forms of message passing seen in the languages discussed in Section 3. At best, VHDL signal assignment might be viewed as asynchronous unbuffered message passing, leading to loss of messages if the receiver is not ready to accept them.

There are a number of issues to consider when designing message-passing communication mechanism in VHDL:

· whether the message send operation should name a target process as the recipient, or a communication channel as the transmission medium,

· whether message passing should be asynchronous or synchronous,

· whether to allow broadcasting of messages, and

· how message passing integrates with concrete signal assignment.

Given that a description may be refined to a hardware implementation in which communication occurs via named signals, use of named communication channels is the appropriate choice for the first issue. It is a more natural abstraction of the communication mechanism used in hardware description. Furthermore, it allows a communicating process to be encapsulated with formal channels. Such a process can then be multiply instantiated, with each instance communicating with different partner processes.

The choice between asynchronous and synchronous message passing is not simple; both forms can be seen in different system-level design languages. On the one hand, asynchronous message passing has the appeal of freeing the designer from ensuring that sending and receiving processes are ready to communicate at the same time. Messages from the sender are simply buffered until the receiver is ready to accept them. Furthermore, asynchronous message passing can be seen as an abstraction of signal assignment, since the latter is also a form of asynchronous communication. However, it is difficult to reason about systems described in terms of asynchronous message passing. Proofs that messages are eventually received must either rely on the assumption of indefinitely large buffering capacity, or take account of some bounded capacity and the possibility of system failure due to buffer overflow. When a system is implemented, if registers of FIFOs are used to implement buffering of communication, equivalence checking must also take account of the finite buffering provided in the implementation.

Synchronous message passing, on the other hand, is more amenable to formal analysis, and properties such as freedom from deadlock and livelock can be expressed using CSP [22], for example, as the underlying mathematical model. Furthermore, synchronous message passing more accurately reflects the desired behaviour of hardware modules that synchronize during communication, for example, using hand-shaking or a common clock without buffering.

Both asynchronous and synchronous message passing can be seen as valid abstractions of communication implemented in hardware and software. Each form can be expressed in terms of the other. For example, asynchronous communication can be expressed in terms of synchronous communication using explicitly instantiated message queues, and synchronous communication can be expressed in terms of asynchronous communication using explicit handshaking. It may be appropriate to include mechanisms for both forms in the language, though this adds complexity. Further research is needed to resolve this issue.

In considering the third issue, the choice of communication via named channels means that broadcasting amounts to multiple processes receiving from a given channel. This parallels hardware communication, in which a signal from one source can be connected to several receivers. If synchronous communication is used, the presence of multiple receivers implies a barrier beyond which none of the receivers nor the sender can pass until message transmission occurs. If asynchronous communication is used, each receiver accepts a copy of the message when it is ready. The sender proceeds as soon as it has sent the message.

In considering the fourth issue, one might attempt to express message passing as form of signal assignment by generalizing the existing signal assignment semantics into a more abstract form. This involves abstracting away from the detailed notions of time and delay associated with signal assignment. Further, the details of editing transaction lists in signal drivers should be removed, and the need for resolution of multiple driving values should be avoided. One way in which these requirements can be met is to provide an abstract form of signal, leading to buffered asynchronous communication. Message sending simply involves queuing a value on the signal, and message reception involves accepting the next value from the queue. This might be expressed in a language extension as a new class of abstract signal. For example, such signals might be used to communicate with a process representing an elevator in a building as follows:

```
signal elevator_call : floor_number abstract;
signal elevator_location : floor_number abstract;

elevator : process is
begin
    . . .
    wait on elevator_call;  -- receive next message
    calling_floor := elevator_call;
    elevator_location <= current_floor; -- send message
    . . .
end process elevator;
```

Given such an extension, it would be necessary to allow specification of abstract signals in interfaces. For example, an entity interface for an operator console might be written as follows:

```
entity operator_console is
    port ( signal status : in status_msg abstract;
        signal command : out command_msg abstract );
end entity operator_console ;
```

Given the significant difference between the semantics of such abstract signals and the concrete signals currently in the language, it may be more appropriate to cast abstract signals as a separate syntactic construct, for example, as communication channels. This approach would also be

preferred if synchronous message passing were adopted. The above examples might be rewritten using a channel construct as follows:

```
channel elevator_call : floor_number;
channel elevator_location : floor_number;

elevator : process is
begin
    . . .
    receive calling floor from elevator_call;
    send current_floor to elevator_location;
    . . .
end process elevator;
```

and

```
entity operator_console is
    port ( channel status : in status_msg;
           channel command : out command_msg );
end entity operator_console ;
```

5.3 Process Abstraction

The model of concurrency in VHDL is based on processes which are statically specified in architecture bodies. However, the language does not allow specification of a process type that can be separately instantiated. Instead, the process must be encapsulated in a design entity and instantiated through the component instantiation mechanism. This is cumbersome, and has the disadvantage of implying structural partitioning.

This deficiency can be overcome by extending VHDL to include process types, abstracting over the statically specified processes currently provided in the language. Processes interact with their environment through the communication mechanism provided by the language, so a process abstraction should include an interface in which formal communication objects can be specified. A process type can be statically instantiated as a concurrent statement within an architecture body, with bindings made between formal and actual communication objects.

As an example, consider a process type representing an abstraction of the elevator described above, communicating using message channels:

```
type elevator is process body
    port ( channel elevator_call : in floor_number;
           channel elevator_location : out floor_number );
begin
    . . .
    receive calling floor from elevator_call;
    send current_floor to elevator_location;
    . . .
end process elevator;
```

This process type might be instantiated to represent multiple elevators in a building:

```
for elevator_number in
        1 to number_of_elevators generate
    an_elevator : process elevator
        port map ( calling_floor => call(elevator_number),
                   elevator_location
                      => location(elevator_number) );
```

5.4 Dynamic Instantiation

Another deficiency in VHDL for system-level modeling is that it does not provide for dynamic creation of processes, preventing it from being used to model systems as dynamically varying collections of active objects. Process types, described in Section 5.3, can be used as the basis for dynamic creation of processes in an extension to VHDL. As the example below shows, channel types and dynamically created message channels may also be needed to communicate with dynamically created processes. The programming language Ada [30] provides a good model for the semantics of process instantiation, activation and termination, since the process types described above are similar to task types in Ada.

An example of a system requiring dynamic process instantiation is a client-server system in which the server is multithreaded, allowing it to serve multiple transactions concurrently. If the number of clients to be served concurrently is not known *a priori*, the server may create agents dynamically to perform the transactions. The organization of the system is illustrated in Figure 1. The system may ultimately be implemented in software, but it desirable to model it early in the design flow before hardware/software partitioning is performed.
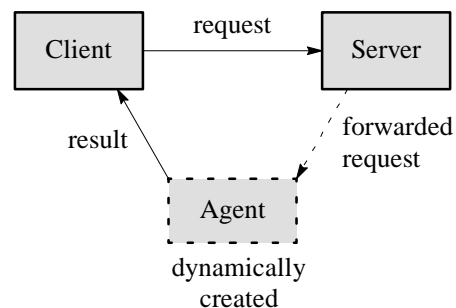
**Figure 1. A client-server system
with dynamically created agents.**

The types representing the message channels between processes are described as follows:

```
type result_value is . . .;
type result_channel is channel result_value;
type result_ref is access result_channel;
```

```
type request_info is record
    . . .;  -- info for the transaction
    result_please : result_ref;
end record request_info;
type request_channel is channel request_info;
type request_ref is access request_channel;
```

The type result_channel represents a channel for receiving result messages from the server, and the type result_ref is a reference to such a channel. The type request_info is the message type for requests to the server. It includes a reference to the channel upon which the client expects to receive the result of the request. The type request_channel represents a channel for sending requests, and the type request_ref is a reference to a request channel.

A client is described by the following process type:

```
type client is process body
    port ( channel request : out request_channel );

    variable result : result_ref := new result_channel;

begin
    . . .
    send ( . . ., result ) to request;
    receive . . . from result.all;
    . . .
end process body client;
```

The client's port is a channel upon which it sends requests. Part of the client's state is a dynamically created channel for receiving transaction results. When the client makes a request, it includes the reference to its result channel as part of the request.

The server is described by the following process type:

```
type server is process body
    port ( channel request : in request_channel );

    type agent is process body
        port ( channel request : in request_channel );

        variable info : request_info;

    begin  -- agent
        receive info from request;
        . . .;  -- perform transaction
        send . . . to info.result_please.all;
        exit;
    end process body agent;

    type agent_ref is access agent;

    variable info : request_info;
    variable new_agent_request : request_ref;
    variable new_agent : agent_ref;

begin  -- server
    receive info from request;
    new_agent_request := new request_channel;
    new_agent := new agent
                port map ( new_agent_request.all );
    send info to agent_request.all;
end process body server;
```

The server has a channel port for receiving requests, and encapsulates a process type for agents, which also has a channel port for requests. The body of the server receives a request message on its request channel, and saves the request in the info variable. It then dynamically creates a new request channel and a new agent process, with the agent's request channel port mapped to the new request channel. The server then forwards the saved request message via the new channel. The newly created agent receives the forwarded message, performs the transaction, and sends the results to the channel referenced in the request message. The agent then terminates. While the agent was processing the transaction, the server may have received further request messages and created agents to process them concurrently.

5.5   Other Issues

One of the main motivations for extending VHDL with system-level modeling features is to allow it to be used early in the design flow, before partitioning between hardware and software implementation is performed. Such extensions improve the useability of the language for performance modeling and for validation of functional requirements. The extensions should integrate with tools and methodologies for hardware/software codesign and synthesis [14, 19, 38], aiding refinement of system-level models to lower-level implementations. Techniques for automatic cosynthesis rely on partitioning the model into execution threads at the process level, at the basic-block level, or at some intermediate level (for example, at the level of sequences between communication operations). The cosynthesis tool partitions based on cost and performance constraints, factoring in the overhead in communicating between hardware and software components, and generates software instruction sequences and hardware modules. This process is assisted by an input language that avoids bias toward either hardware or software implementation, and that can be analyzed to determine processes, communication operations and basic blocks. The process and communication models discussed in previous sections meet this requirement.

Extensions for system-level modeling should not be added as an isolated aspect of the language. They should be closely integrated with other semantic mechanisms, and preserve the underlying design philosophies of the language. Furthermore, system-level modeling extensions must integrate with other extensions, such as those under study within IEEE working groups for object-oriented language features and for interface specification.

6.   Conclusion

In this paper we have identified issues that must be addressed by system-level description languages, and focussed on issues for extending VHDL for system-level

behavioural and structural modeling. Such extensions would allow VHDL to be used early in the design flow, where informal methods are often used today. Use of more rigorous modeling methods allows validation of requirements through simulation and formal analysis, and allows refinement of the system design to be verified.

The SUAVE Project [5], has already defined extensions for adding both object-oriented and genericity features to VHDL, improving the language's expression of data abstractions. Implementation work on these extensions is in progress. Further work in SUAVE will address extensions to the concurrency and communication models, based on the analysis of issues presented in this paper.

References

[1] P. Alexander and P. Baraona, "Extending VHDL to the System Level," *Proceedings of VHDL International Users' Forum Fall 1997 Conference*, Arlington, VA, pp. 96–104, 1997.

[2] P. J. Ashenden, *The Designer's Guide to VHDL*. San Francisco, CA: Morgan Kaufmann, 1996.

[3] P. J. Ashenden and P. A. Wilsey, *A Comparison of Alternative Extensions for Data Modeling in VHDL*, Dept. Computer Science, University of Adelaide, Technical Report TR-02/97, ftp://ftp.cs.adelaide.edu.au/pub/VHDL/TR-data-mo deling.ps, 1997.

[4] P. J. Ashenden and P. A. Wilsey, "Considerations on Object-Oriented Extensions to VHDL," *Proceedings of VHDL International Users Forum Spring 1997 Conference*, Santa Clara, CA, pp. 109–118, 1997.

[5] P. J. Ashenden and P. A. Wilsey, *SUAVE: A Proposal for Extensions to VHDL for High-Level Modeling*, Dept. Computer Science, University of Adelaide, Technical Report TR-97-07, ftp://ftp.cs.adelaide.edu.au/pub/VHDL/TR-extensio ns.pdf, 1997.

[6] P. J. Ashenden and P. A. Wilsey, "SUAVE: Painless Extension for an Object-Oriented VHDL," *Proceedings of VHDL International Users Forum Fall 1997 Conference*, Washington, DC, 1997.

[7] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, vol. 21, no. 3, pp. 261–322, 1989.

[8] J. Benzakki and B. Djaffri, "Object Oriented Extensions to VHDL: the LaMI Proposal," *Proceedings of Conference on Hardware Description Languages '97*, Toledo, Spain, pp. 334–347, 1997.

[9] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, , pp. 25–59, 1987.

[10] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummins, 1994.

[11] S. Budkowski and P. Dembinski, "An Introduction to Estelle: A Specification Language for Distributed Systems," *Computer Networks and ISDN Systems*, vol. 14, no. 1, pp. 3–23, 1987.

[12] D. Cabanis and S. Medhat, "Classification-Orientation for VHDL: A Specification," *Proceedings of VHDL International Users Forum Spring '96 Conference*, Santa Clara, CA, pp. 265–274, 1996.

[13] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, vol. 6. New York, NY: Springer-Verlag, 1985.

[14] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Co-synthesis for Microcontrollers," *IEEE Design and Test of Computers*, vol. 10, no. 4, pp. 64–75, 1993.

[15] A. S. Evans, "Specifying and Verifying Concurrent Systems Using Z," *Proceedings of FME '94: Industrial Benefit of Formal Methods*, pp. 366–380, 1994.

[16] O. Færgemand and A. Olsen, "Introduction to SDL-92," *Computer Networks and ISDN Systems*, vol. 26, , pp. 1143–1167, 1994.

[17] M. Fowler and K. Scott, *UML Distilled*. Reading, MA: Addison-Wesley, 1997.

[18] J. Gong, D. D. Gajski, and S. Bakshi, "Model Refinement for Hardware-Software Codesign," *ACM Tansactions on Design Automation of Electronic Systems*, vol. 2, no. 1, pp. 22–41, 1997.

[19] R. Gupta and G. de Micheli, "System Co-synthesis for Digital Systems," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 29–41, 1993.

[20] D. Harel, "Statecharts: A Visual Formalism for Computer Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[21] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 11, pp. 934–941, 1978.

[22] C. A. R. Hoare, *Communicating Sequential Processes*. London: Prentice Hall, 1985.

[23] Y. V. Hoskote, J. A. Abraham, D. S. Fussell, and J. Moondanos, "Automatic Verification of Implementations of Large Circuits Against HDL Specifications," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 16, no. 3, pp. 217–228, 1997.

[24] ICL, *VHDL+: Extensions to VHDL for System Specification, Version 2.0*. ICL, Language Reference Manual, 1997.

[25] IEEE, *Standard VHDL Language Reference Manual*. Standard 1076-1993, New York, NY: IEEE, 1993.

[26] IEEE, *Standard Verilog Hardware Description Language Reference Manual*. Standard 1364-1995, New York, NY: IEEE, 1995.

[27] INMOS Ltd., *OCCAM Programming Manual*. London, UK: Prentice-Hall, 1984.

[28] International Standards Organization, *Information processing systems—open systems interconnection. {LOTOS}: a formal description technique based on the temporal ordering of observational behaviour*. International Standard 8807, 1988.

[29] ISO, *Estelle: A Formal Description Technique Based on an Extended State Transition Model*. Draft International Standard 9074, 1987.

[30] ISO/IEC, *Ada 95 Reference Manual*. International Standard ISO/IEC 8652:1995 (E), Berlin, Germany: Springer-Verlag, 1995.

[31] ITU, *Specification and Description Language (SDL)*. Revised Recommendation Z.100, 1992.

[32] Y.-L. Lin, "Recent Developments in High-Level Synthesis," *ACM Tansactions on Design Automation of Electronic Systems*, vol. 2, no. 1, pp. 2–21, 1997.

[33] M. C. McFarland, "Formal Verification of Sequential Hardware: A Tutorial," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 12, no. 5, pp. 633–654, 1993.

[34] M. Radetzki, W. Putzke, W. Nebel, S. Maginot, J.-M. Bergé, and A.-M. Tagant, "VHDL Language Extensions to Support Abstraction and Re-Use," *Proceedings of Workshop on Libraries, Component Modeling, and Quality Assurance*, Toledo, Spain, 1997.

[35] Rational Software Corporation, *UML Resource Center*. Rational Software Corporation, http://www.rational.com/uml/, 1997.

[36] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. New York, NY: Prentice-Hall, 1992.

[37] S. Swamy, A. Molin, and B. Covnot, "OO-VHDL: Object-Oriented Extensions to VHDL," *IEEE Computer*, vol. 28, no. 10, pp. 18–26, 1995.

[38] D. E. Thomas, J. K. Adams, and H. Schmit, "A Model and Methodology for Hardware-Software Codesign," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 6–15, 1993.

[39] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, Third ed. Boston, MA: Kluwer Academic Publishers, 1996.