Accepted for publication in *Proceedings of Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98).*



# Extensions to VHDL for Abstraction of Concurrency and Communication*

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 describes extensions to VHDL to support system-level behavioral modeling by providing more abstract forms of communication and concurrency than those currently in the language. The report summarizes design objectives and issues that must be considered in developing such extensions, and presents definitions of our extensions. The extensions for communication consist of channel types, channel objects, dynamically allocated channels, and message passing statements. The extensions for concurrency consist of process declarations and static and dynamic process instantiation statements. Use of the extensions is illustrated with examples.*

## 1. Introduction

As the complexity of integrated hardware and software systems increases, system-level design languages are becoming increasingly important. Such languages rely on abstraction as the key to managing complexity. 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. The domains under consideration include behavior, structure, performance, physical arrangement and packaging, power consumption, thermal, cost, and so on. 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.

Hardware description languages focus on describing systems in the behavioral and structural domains. However, due to their origin as languages for hardware design, they frequently do not include strong capabilities for abstracting over data and for describing complex interactions. For example, in Verilog [14, 18], data types are closely bound to their binary representation, and signalling between modules includes aspects of electrical implementation. VHDL [1, 13], on the other hand, allows more abstract expression of data, and its type system is similar to that of conventional programming languages. However, its signalling features are still closely bound to electrical implementation.

To remedy these deficiencies, we have developed extensions to VHDL to improve its support for system-level modeling. These extensions are based on the requirement in a system-level description language for abstraction in the following areas:

- abstraction of data,
- abstraction of concurrency, and
- abstraction of communication and timing.

These extensions make VHDL suitable for describing at an abstract level aspects of a system that may ultimately be implemented in hardware or software. We have described the details of the extensions to the data modeling facilities in previous papers [4–6]. These extensions involve mechanisms for object-oriented data types and for genericity. We have also presented a discussion of the issues that must be considered in extending VHDL to provide more abstract forms of concurrency and communication [3].

* This work was partially supported by Wright Laboratory under USAF contract F33615-95-C-1638.

In this paper we present our extensions for abstraction of communication and concurrency in the SAUVE (SAVANT and University of Adelaide VHDL Extensions) project. We introduce into VHDL the notions of communication channels and message-passing operations as an abstraction of communication by signals. We also extend the process model by allowing process declarations that can be statically or dynamically instantiated.

Section 2 of this paper reviews our design objectives. Section 3 discusses issues that must be considered in extending VHDL with a more abstract form of communication and gives reasons for our choices among the alternaitves. Section 4 presents the details of the abstract communication language features in SUAVE and Section 5 presents the details of the extensions for concurrency abstraction. Section 6 presents an extended example, a multithreaded client-server system, that illustrates the combined use of the extensions in a system-level model. Finally, Section 7 contains our conclusions.

## 2. Design Objectives

Our main design objective in the SUAVE project is to improve high-level modeling support in VHDL through increased use of abstraction. Specific objectives leading to the extensions described in this paper are:

- to provide a more abstract form of communication than the existing mechanisms of signals and signal assignment,
- to provide dynamic process creation and termination,
- to provide abstractions that are not biased towards hardware or software implementations, allowing subsequent partitioning and refinement (hardware/software co-design),
- to preserve capabilities for synthesis and other forms of design analysis,
- to ensure clean integration and well-defined interaction with existing language mechanisms,
- to ensure clean integration and well-defined interaction with extensions for data modeling and genericity developed in the SUAVE Project, and
- to preserve correctness of existing models within the extended language.

We consider integration with the existing language to be a key design objective. We are guided by Fred Brooks' notion of "conceptual integrity" [8]. As Brooks notes, "Conceptual integrity does require that a system reflect a single philosophy and that the specification as seen by the user flow from a few minds." To this end, we have embraced the design principles (listed in our earlier paper [5]) used during earlier development and standardization of VHDL.

## 3. Considerations for the Abstraction of Communication

At the system level of design, 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 process. The act of message passing can also be used to synchronize processes. SUAVE extends VHDL with message passing for abstract communication as it is 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. Fortunately, they are easily expressed in terms of message passing.

There are two ways that message passing abstracts away 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 is therefore 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.

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 other system-level description languages such as Estelle [9, 15], SDL [10, 16] and CSP [11, 12]. 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.

In a previous paper [3], we identify a number of issues to consider when designing message-passing communication mechanism in VHDL and discuss some of the alternatives. The issues are:

1. whether the message send operation should name a target process as the recipient, or a communication channel as the transmission medium;
2. whether message passing should be asynchronous or synchronous;
3. whether to allow multicasting of messages; and
4. how message passing integrates with concrete signal assignment.

Our choices among the alternatives are as follows.

For the first issue, given that a description may be refined to a hardware implementation in which communica-

tion occurs via named signals, named communication channels are most appropriate. Channels are a more natural abstraction of the communication mechanism used in hardware description. Furthermore, they allow a communicating process to be encapsulated with formal channels. Such a process can then be instantiated several times, each instance communicating with different partner processes.

For the second issue, SUAVE chooses asynchronous message passing. While either form of can be used to implement the other, asynchronous message passing is the most flexible. Synchronous communication can be simply expressed using handshaking. The details can be encapsulated to provide the appearance of simple synchronous message passing, rendezvous, or remote procedure call. Implementing asynchronous communication with synchronous primitives, on the other hand, requires explicit instantiation of a message buffer. An additional consideration addresses correctness proofs for communicating programs. While formal proof techniques for synchronous communication may be simpler, techniques for proving properties of asynchronous communication have been developed [17].

For the third issue, SUAVE allows multiple processes to receive from a channel, thus implementing a form of multicast communication. Each receiver accepts a copy of the message when it is ready. The sender proceeds as soon as it has sent the message. This parallels hardware communication, in which a signal from one source can be connected to several receivers.

For the fourth issue, the previous paper identified two alternatives: (i) generalizing signals and signal assignment to a more abstract form, and (ii) adding channels as a new language construct. While the former alternative is possible, in practice it is difficult to define. To do so involves adding numerous special-case rules to the semantic definitions of signal declarations, interface signals, signal assignment statements and wait statements. Adding channels is easier to define, and, since the semantics are sufficiently different from signals, easier to comprehend. Hence, SUAVE follows the latter approach.

# 4. Channels and Communication in SUAVE

## 4.1 Channels

Abstract communication in SUAVE occurs over *channels*, which are of declared *channel types*. Channels can be declared objects or interface objects. The syntax rule for a channel type definition is:

channel_type_definition ::=
    **channel of** subtype_indication
  | **null channel**

In the first form of channel type definition, the subtype indication is called the *message type* of the channel. It denotes the subtype of values that may be passed as messages on a channel of the channel type. The second form of channel type definition defines a *null channel type*. Such a channel type is used for a channel on which the messages have no data content.

One or more channels may be declared using a channel declaration. The syntax rule is:

channel_declaration ::=
    **channel** identifier_list : subtype_indication ;

Channel declarations may appear within entity declarations, architecture bodies, block statements, generate statements, and package declarations. The subtype indication in the channel declaration denotes a channel type.

A channel is analogous to a signal, except that information is transferred using the *send* and *receive* message passing operations (described below). There is no notion of resolution of multiple source values, nor of specific times at which values occur on channels. A channel object denotes a queue of *messages*. When the channel object is created, the queue is initially empty.

Example

The following declarations define three channel types and two channel objects:

```
type request_channel is channel of request_message;
type result_channel is channel of result_message;

type acknowledgment_channel is null channel;

channel request : request_channel;
channel result : result_channel;
```

——

SUAVE also allows *interface channels*, which may appear as formal ports of design entities, components or blocks, or as formal channel parameters of subprograms. The syntax rule is:

interface_channel_declaration ::=
    **channel** identifier_list : [ mode ] subtype_indication

The mode, if present, is one of **in** or **out**, and the subtype indication denotes a channel type. An **in** mode channel may be used to receive messages, and an **out** mode channel may be used to send messages.

Example

In the following architecture body, the image_channel type represents tokens in an uninterpreted queuing model. The component image_filter has channel ports for receiving and sending tokens. The component instance filter has its ports associated with the actual channel objects raw_image and filtered_image.

```
architecture performance_modeling of motion_detector is
    type image_channel is channel of image_token;
```

```
    component image_filter is
        port ( channel raw_image : in image_channel;
               channel filtered_image : out image_channel );
    end component image_filter;

    channel raw_image, filtered_image : image_channel;
    . . .
begin
    filter : component image_filter
        port map ( raw_image => raw_image,
                   filtered_image => filtered_image );
    . . .

end architecture performance_modeling;
```

---

## 4.2 Communication Statements

SUAVE extends the set of sequential statements to include *send statements*, *receive statements* and *select statements*. A send statement adds a message to the queue of a channel. The syntax rule is:

```
send_statement ::=
        [ label : ] send [ expression ] to channel_name ;
```

The expression is disallowed if the channel is of a null channel type. Otherwise, the expression is required and denotes the value to be sent as a message. If the channel is of a null channel type, a data-less message is sent. Execution of a send statement involves adding the message to the tail of the message queue of the named channel. The process executing the send statement then continues executing. If multiple processes execute send statements to the same channel concurrently, the order in which the messages are added to the message queue is not defined. (It is implementation dependent.)

Example

The following two statements send (a) to a channel with data and (b) to a null channel:

```
send result_message'( . . . ) to result;
send to acknowledgment;
```

---

A process accepts a message from a channel using a *receive statement*. The syntax rule is:

```
receive_statement ::=
        [ label : ] receive [ target ] from channel_name ;
```

The target is disallowed if the channel is of a null channel type, otherwise it is required. The target must denote a variable name or an aggregate of variable names. Execution of a receive statement involves examining the message queue of the named channel. If the message queue is empty, the process suspends until a message arrives. When there is a message available, it is removed from the queue. If the channel is not of a null channel type, the value of the message is assigned to the target using the same rules as variable assignment.

If multiple processes can read a message channel, all processes receive each message sent to the channel. Furthermore, all processes receive the messages from the channel in the same order. An implementation may achieve this effect either by providing one message queue for the channel, from which each process copies message values, or by replicating the message queue at each process.

Example

The following two statements receive (a) from a channel with data and (b) from a null channel:

```
receive next_request from request;
receive from acknowledgment;
```

---

A process may choose between a number of channels for message reception using a *select statement*. The syntax rules are:

```
select_statement ::=
        [ select_label : ]
                select
                        [ guard ] receive_alternative
                { or
                        [ guard ] receive_alternative }
                | else
                        sequence_of_statements ]
                end select [ select_label ] ;

guard ::= when condition =>

receive_alternative ::=
        receive_statement [ sequence_of_statements ]
```

A select statement allows non-deterministic choice between alternative sources for message reception. Each receive alternative may be guarded by a boolean condition; a guarded alternative may only be chosen if the guard is true.

Execution of the select statement consists firstly of evaluating the guard conditions. An alternative is said to be *open* if it has no guard, or if its guard evaluates to true. If no alternative is open and the select statement has an **else** clause, the statements in the **else** clause are executed, thus completing execution of the select statement. It is an error if no alternative is open and there is no **else** clause.

If there are open alternatives for which the channels named in the corresponding receive statements have queued messages, one of the open alternatives is chosen arbitrarily. The receive statement is executed, followed by execution of the sequence of statements (if present), completing execution of the select statement.

If there are open alternatives but none of the channels named in the corresponding receive statements have

queued messages, execution depends on whether the select statement has an **else** clause. If there is an **else** clause, the statements in it are executed, completing execution of the select statement. Otherwise, the process blocks until a message arrives on one of the channels named in the receive statements of the open alternatives. Execution then proceeds as described in the previous paragraph. The guard conditions are not re-evaluated while the process is blocked or when a message arrives.

Example

In the following example, the process access_controller arbitrates between readers and writers of a shared resource. A reader sends a read-request message to the process, and only proceeds when the process responds with an acknowledgment. When the reader finishes reading, it sends a read-finished message to the process. Writers obey a similar protocol. Multiple readers are allowed concurrent access, provided the number of active writers is zero. Only one writer at a time is permitted, and then only if there are no active readers. The guards in the select statement control the reception of request messages, based on the number of readers or writers currently active.

```
type read_request_channel is channel of . . . ;
type read_finished_channel is null channel;
type write_request_channel is channel of . . . ;
type write_finished_channel is null channel;
. . .

channel read_request : read_request_channel;
channel read_finished : read_finished_channel;
channel write_request : write_request_channel;
channel write_finished : write_finished_channel;
. . .

access_controller : process is
    variable number_of_readers, number_of_writers : natural := 0;
begin
    select
        when number_of_writers = 0 =>
            receive read_request_info from read_request;
            number_of_readers := number_of_readers + 1;
            . . . – – acknowledge read request
    or
        receive from read_finished;
        number_of_readers := number_of_readers – 1;
    or
        when number_of_readers = 0
                and number_of_writers = 0 =>
            receive write_request_info from write_request;
            number_of_writers := number_of_writers + 1;
            . . . – – acknowledge write request
    or
        receive from write_finished;
        number_of_writers := number_of_writers – 1;
    end select;
end process access_controller;
```

———

### 4.3 Dynamically Created Channels

SUAVE provides mechanisms based on access types for dynamically creating channels in order to communicate with dynamically created processes. An access types may be declared to have a channel type as its designated type. Such an access type is called an *access-to-channel* type. A channel may be dynamically allocated using an allocator with a subtype indication denoting a channel type. The access value returned by the allocator designates the newly allocated channel.

Example

The following declarations define result_ref to be an access-to-channel type, and the variable result to be of this type, initialized with a reference to a dynamically created channel.

```
type result_ref is access result_channel;
variable result : result_ref := new result_channel;
```

———

## 5. Extensions for Abstraction of Concurrency

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.

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. Furthermore, it does not allow dynamic instantiation of processes.

These deficiencies can be overcome by extending VHDL to include process declarations, abstracting over the statically specified processes currently provided in the language. A process interacts with its environment using the communication mechanism provided by the language. Therefore, a process declaration includes an interface in which formal communication objects can be specified. A process declaration can be statically instantiated as a concurrent statement in an architecture body, with bindings made between formal and actual communication objects.

It can also be dynamically instantiated by the execution of a sequential process instantiation statement. Process declarations and their instantiation and termination are described more fully below.

## 5.1 Process Declarations

SUAVE extends declarative parts to include process declarations and process bodies as follows:

```
process_declaration ::=
        process_specification
        end process [ process_simple_name ] ;

process_body ::=
        process_specification
                process_declarative_part
        begin
                process_statement_part
        end process [ process_simple_name ] ;

process_specification ::=
        process identifier is
                [ generic_clause ]
                [ port_clause ]
```

Process declarations, like subprogram declarations, may be defined with separate specifications and bodies. In particular, if a process is declared in a package, the process specification occurs in the package declaration, and the process body occurs in the package body.

## 5.2 Process Instantiation Statements

Static instantiation of a declared process is done using a *process instantiation statement*. The syntax rule is:

```
process_instantiation_statement ::=
        [ instantiation_label : ]
                process process_name
                        [ generic_map_aspect ]
                        [ port_map_aspect ] ;
```

A process instantiation statement is equivalent to a block statement with the generic clause and port clause taken from the process specification and the generic map aspect and port map aspect taken from the process instantiation statement. The declarative part of the block statement is empty, and the statement part contains a process whose declarative part and statement part are taken from the process body. The meaning of any identifier within the block statement and the process it contains is that associated with the identifier in the process declaration or body. To illustrate application of these rules, consider the following process body and instantiation statement:

```
process p is
    generic ( g : integer );
    port ( channel c : c_chan );
    variable v : integer;
begin
    v := x;
end process p;
. . .
p_inst : process p
    generic map ( g => 5 )
    port map ( c => c1 );
```

The process instantiation statement is semantically equivalent to:

```
p_inst : block is
    generic ( g : integer );
    generic map ( g => 5 );
    port ( channel c : c_chan );
    port map ( c => c1 );
begin
    p : process is
        variable v : integer;
    begin
        v := ... . x;
    end process p;
end block p_inst;
```

The name x is prefixed to ensure that it refers to the same item visible in the process declration rather than any homograph that hides the name.

Dynamic instantiation of a process is performed using a *sequential process instantiation statement*. The syntax rule is:

```
sequential_process_instantiation_statement ::=
        [ label : ]
                process process_name
                        [ generic_map_aspect ]
                        [ port_map_aspect ] ;
```

Execution of a sequential process instantiation statement involves the following steps:

- elaboration of the generic list of the process declaration to create the formal generics, and association of the actual generics with the formal generics;
- elaboration of the port list of the process declaration to create the formal ports, and association of the actual signals, channels and values with the formal ports;
- elaboration of the declarations of the process; and
- creation and initialization of the drivers of the process.

The newly instantiated process then commences execution of its statement part concurrently with the instantiating process in the current simulation cycle. The newly instantiated process is said to *depend* on the instance or activation of the declaration or statement that immediately contains the declaration of the process. That instance or activation may not return or terminate until all of the processes that depend on it have terminated, since such processes may refer to items declared by the declaration or statement.

## 5.3 Process Termination

A process may terminate by executing a sequential statement called a *terminate statement*. The syntax rule is:

terminate_statement ::=
    [ label : ] **terminate** ;

Termination of a process involves the following actions:

- The process waits until all processes that depend on it have terminated.
- The drivers of the process are disconnected from the signals that they drive.
- The formal ports are disassociated from the actual signals and channels.

## 6. Example: A Client-Server System

This example is a model of a client-server system in which the server is multi-threaded, allowing it to serve multiple transactions concurrently. Since the number of clients to be served concurrently is not known *a priori*, the server creates 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 is desirable to model it early in the design flow before hardware/software partitioning is performed.

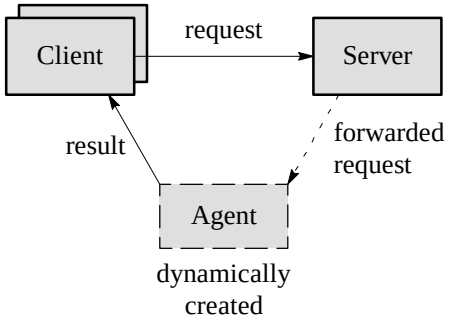


**Figure 1. A client-server system with dynamically created agents.**

The 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.

The client process'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 process has a channel port for receiving requests, and encapsulates a process declaration 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 variable info. 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 is processing the transaction, the server may receive further request messages and create agents to process them concurrently.

```
architecture system_level of client_server_system is

    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;


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

        variable result : result_ref := new result_channel;

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


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

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

            variable info : request_info;

        begin
            receive info from request;
            . . .; – – perform transaction
            send . . . to info.result_please.all;
            terminate;
        end process agent;

        variable info : request_info;
        variable new_agent_request : request_ref;

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


    channel server_request : request_info;

begin

    the_server : process server
        port map ( request => server_request );
```

```
    client_pool : for client_index in 1 to 10 generate
        a_client : process client
            port map ( request => server_request );
    end generate client_pool;

end architecture system_level;
```

## 7. Conclusion

Design at the system level relies on abstraction to manage complexity. In this paper, we have described extensions to VHDL that introduce abstract forms of communication and concurrency. These extensions make the language suitable for design of behavior and structure at the system level. Our extensions are not biased towards hardware or software refinement of a design. Thus, the extended language can be used to express behavior and structure of a system before partitioning into hardware and software, supporting exploration of hardware/software trade-offs and hardware/software co-design. The approach we have taken is to provide abstract forms of the existing language mechanisms for communication and concurrency. This eases refinement of hardware partitions of a design to lower-level implementations expressed in VHDL. The abstract forms of communication and concurrency also ease refinement of the software partitions to programming-language implementation.

Whereas this paper provides an overview of the language extensions, a more detailed specification can be found in a separate Technical Report [2]. Subsequent work in the SUAVE project will involve implementing the extensions in the SAVANT framework [19], and validating the language design with use cases to be published by the $SI^2$ System Level Design Language (SLDL) Committee.

## References


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

[2] P. J. Ashenden and P. A. Wilsey, *Proposed Extensions to VHDL for Abstraction of Concurrency and Communication*, Dept. Computer Science, University of Adelaide, Technical Report TR-97-11, 1997.

[3] P. J. Ashenden and P. A. Wilsey, "Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL," *Proceedings of VHDL International Users Forum Spring 1998 Conference*, Santa Clara, CA, pp. 42–50, 1998.

[4] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "Reuse Through Genericity in SUAVE," *Proceedings of VHDL International Users Forum Fall 1997 Conference*, Arlington, VA, pp. 170–177, 1997.

[5] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, *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-extensions.pdf, 1997.

[6] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, "SUAVE: Painless Extension for an Object-Oriented VHDL," *Proceedings of VHDL International Users Forum Fall 1997 Conference*, Arlington, VA, pp. 60–67, 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] F. P. Brooks, Jr., *The Mythical Man-Month*, Anniversary ed. Reading, MA: Addison-Wesley, 1995.

[9] 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.

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

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

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

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

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

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

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

[17] R. D. Schlichting and F. B. Schneider, "Understanding and Using Asynchronous Message-Passing," *Proceedings of 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, pp. 141–147, 1982.

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

[19] P. A. Wilsey, D. E. Martin, and K. Subramani, "SAVANT/TyVIS/warped: Components for the Analysis and Simulation of VHDL," *Proceedings of VHDL International User's Forum Spring 1998 Conference*, Santa Clara, CA, pp. 195–201, 1998.