Accepted for publication in *Proceedings of the 1999 International Hardware Description Languages Conference and Exhibit (HDLCON '99).*

Communication and Synchronization Using Bounded Channels in SUAVE*

Peter J. Ashenden and Robert Esser
Department of Computer Science
University of Adelaide, SA 5005
Australia
petera@cs.adelaide.edu.au
esser@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 described improvements to the abstract interprocess communication features added to VHDL as part of the SUAVE language design. Channel type declarations are extended to allow specification of bounded message buffers. This allows the designer to choose between asynchronous or synchronous message passing semantics. The latter makes models more amenable to formal verification based on state-space exploration. The language is also extended to allow specification of timeout intervals in select statements. This change makes the language more widely applicable, including for description of telecommunication protocols.

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 [12, 18], data types are closely bound to their binary representation, and signalling between modules includes aspects of electrical implementation. VHDL [1, 11], 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 [2, 4–7]. These extensions are based on the requirement in a system-level description language for abstraction of data, concurrency, communication and timing. The mechanisms added to the language to support abstraction in these areas include:

- object-oriented data types,
- type generics,
- process declarations, with static and dynamic instantiation, and
- message-passing communication on channels.

This paper focuses on the last of these mechanisms. The communications mechanisms originally proposed provided asynchronous message passing on channel with unbounded buffer capacity. While this is appropriate for some modeling applications, other applications find synchronous communication or communication with bounded buffering more appropriate. Furthermore, formal proof of properties and refinements of communicating system are more readily achieved when synchronous communication or bounded buffering is used. Another deficiency in the original proposal was the lack of a mechanism to specify timeouts in communication statements. Such mechanisms

* This work was partially supported by a grant from Motorola Australia Software Centre.

are widely used in description of communication protocols, and its omission from SUAVE reduces the language's applicability in that area.

In this paper, we generalize the communication mechanisms of SUAVE to allow specification of buffer capacity of channels and specification of timeouts in communication statements. Section 2 reviews the communication mechanisms in the previous proposal and discusses the problems that arise. Section 3 presents revised mechanisms for describing channels, including mechanisms for specifying buffering capacity, and discusses the implications for formal proof of model properties. Section 4 presents revised communication statements, including a mechanism for specifying timeouts, and compares the mechanism with features of SDL [13]. We summarize and draw conclusions in Section 5.

2. SUAVE's Existing Features for Abstract Communication

In a previous paper [3], we identified a number of issues influencing our design of communication mechanisms in SUAVE. Based on a consideration of these issues, we chose asynchronous message passing as the communication paradigm. Messages are transferred between processes over named, typed communication channels. Channels may be connected to multiple receiving processes, allowing a form of multicast communication.

2.1 Channel types and objects

A model that uses channels must first declare the necessary *channel types* using channel type declarations. The syntax rule is

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

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 first-in/first-out buffer of *messages*. When the channel object is created, the buffer is initially empty.

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.

SUAVE provides mechanisms based on access types for dynamically creating channels in order to communicate with dynamically created processes. An access type 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.

2.2 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 buffer 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 buffer 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 buffer is not defined. (It is implementation dependent.)

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 buffer of the named channel. If the message buffer is empty, the process suspends until a message arrives. When there is a message available, it is removed from the buffer. 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 buffer for the channel, from which each process copies message values, or by replicating the message buffer at each process.

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 buffered 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 buffered 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.

2.3 Deficiencies in the Existing Features

While our previous proposal was based on an analysis of issues affecting design of communication mechanisms, there are two areas in which improvements can be made. The first relates to the specification of unbounded buffering in channels. Our decision to provide unbounded buffering was influenced by our survey of previous system-level description languages, a number of which provided unbounded buffering (SDL included). Furthermore, we observed that bounded buffering and synchronous communication could be expressed with unbounded-buffered communication using a combination of buffer components and handshaking synchronization. However, it appears that this decision has adverse consequences for formal verification of models. The main problem is that the state space of a model that uses unbounded buffers is potentially infinite. An important class of formal verification techniques is based on state-space exploration [10, 15, 17], including those embodied in automatic verification tools [9, 14, 16]; verification using these tools is intractable when "state-space explosion" occurs. Hence it is desirable to allow specification of buffer bounds for channels so that formal verification techniques and tools can be brought to bear on complex and safety-critical designs.

The second area in which our previous proposal can be improved relates to provision of mechanisms for specifying timeouts on communication statements. Many communication protocol specifications include timeouts as a means of dealing with unreliable communication media. For example, if a receiver does not receive an expected message within a specified interval, it ceases to wait for the message and takes some recovery action. Languages such as SDL, used to specify communications protocols, include mechanisms for describing communication with timeouts. Since description of communications protocols is an important application area, it is desirable to include a timeout mechanism in the communication features of SUAVE.

In order to illustrate these mechanisms, we present here an overview of the techniques used in SDL. Systems are represented in SDL as asynchronously communicating, extended finite state machines. Each extended finite state machine (SDL process) contains a single input buffer of unbounded capacity into which all messages from all input signal routes from other processes are received. Message input is always the first statement of a state transition, and may be followed by tasks that can change the state of local variables, decision nodes in which different actions can be executed depending on the current state, and output statements where messages are sent via output signal routes to other processes.

In a particular SDL state, a process may attempt to receive a number of different messages. Since messages are received from the process's input buffer, they are received (unless explicitly changed) in the order in which they ar-

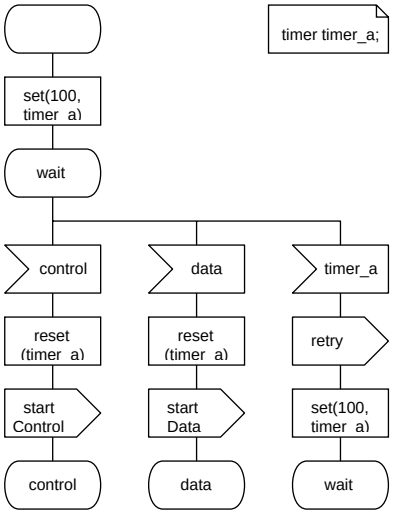

Figure 1. An SDL process using timers for message timeout.

rive. Messages are consumed and ignored if, in a particular state, they are not explicitly defined as being inputs of the that state or defined to be 'saved.'

A timer in SDL is defined to be local to a process. It also acts as a process, in that when it expires it sends a message to the enclosing process's input buffer. The timer can be set and reset by task statements. A side effect of the reset statement is that the enclosing process's input buffer is purged of all expiry messages to ensure that no expiry message can be subsequently received.

Message timeout in a particular state is thus implemented by defining a timer and setting it to a particular timeout value in all state transitions that end in the state. If a message arrives at the input buffer before the timer has expired, the message will be received by the process, which may then reset the timer. Otherwise a timer-expiry message will be generated, causing the transition in which the timer expiry message is received.

Figure 1 shows an example of an SDL process that uses a timer for message timeout. In the initial state, the process sets a timer called timer_a, then enters the state wait to await message reception. If the process receives a data or control message, the process resets the timer and sends the appropriate output message. If, on the other hand, the process receives a timer expiry message from timer_a, the process sends a retry message, sets the timer again, and returns to the wait state.

3. Bounded Channels

In order to meet the need for specifying bounds on sizes of channel message buffers, we have revised the way in which channels are declared in SUAVE. We considered two approaches: allowing specification of buffer size in a channel type declaration, and allowing specification of buffer size in individual channel object declarations.

The first approach, which we chose for our language design, involves all channels of a given type having the same buffer size. If different channels carrying the same message type must have different buffer sizes, they must be declared to be of different channel types.

The alternative approach, which we rejected, allows different channels of the same type to have different buffer sizes. However, this introduces complications in association of actual channels with formal channels of process instances or component instances. If a buffer size is specified for a formal channel, it may conflict with a different size specified for the associated actual channel. Any properties of the process or component inferred from its formal channel size may be invalidated by associating with the formal channel an actual channel with different buffer size. In particular, proofs relating to blocking and synchronization at communication statements may be invalidated. One possible solution to this problem is to require that the actual channel have the same buffer size as the formal channel, but this is effectively the same as the simpler approach of specifying the buffer size in the channel type declaration. Another possible solution is to prohibit specification of buffer size in the formal channel declaration and to derive the size from the actual channel. However, this removes opportunities for proof of properties of the process or component.

Given our choice to specify the buffer size in the channel type, we then considered possible interaction with the existing VHDL mechanism of type constraints. It appeared desirable to allow unconstrained channel types whose buffer sizes are not determined, and to use constraints to specify actual buffer sizes of channel subtypes. This mechanism would allow definition of formal channel ports whose buffer sizes are determined either from generic constants or from the buffer sizes of the associated actual channel objects.

Our initial attempt to design language features based on this approach used unconstrained channel types to denote channels with unbounded buffers, and to allow declaration of unconstrained channel objects. However, this caused two problems. The first problem was an inconsistency with other unconstrained and constrained types, such as array types. VHDL prohibits declaration of objects of other unconstrained types, since the storage to be allocated to such an object cannot be determined when the object is elaborated. The second problem arose from the rules for determining constraints for interface channels. Currently in VHDL, when a constrained actual object is associated with an unconstrained formal object, the formal object takes on the constraints of the actual object. In the case of interface channel objects, this would mean that a formal channel with an unbounded buffer would take on the buffer bounds of the actual channel. The implication for the process send-

ing to the formal channel would be that, whereas it would appear that the process could not block on a send statement (due to the unbounded buffering specified in the formal channel type), the process might in fact block (due to the bounded buffering of the actual channel type). This contradiction of possible inferences about process behavior is undesirable.

The key to resolving these problems was the realization that unbounded and bounded channels are significantly different in nature, distinguished by the potential of a sender to block (as discussed in Section 3.1), and that constraints should only apply to bounded channels. Hence, the approach we settled on was to allow specification of three classes of channel types: unbounded, unconstrained bounded, and constrained bounded. The existing VHDL constraint mechanisms are applicable in the last two classes. We took care in the language design to avoid introducing exceptions to the constraint and subtype rules and to provide mechanisms that allow modular and compositional reasoning about process behavior.

The revised syntax rules covering channel types are:

channel_type_definition ::=
    unbounded_channel_definition
  | unconstrained_bounded_channel_definition
  | constrained_bounded_channel_definition

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

unconstrained_bounded_channel_definition ::=
    channel buffer <> **of** subtype_indication
  | **null channel buffer** <>

constrained_bounded_channel_definition ::=
    channel buffer_constraint **of** subtype_indication
  | **null channel** buffer_constraint

buffer_constraint::=
  buffer simple_expression

The existing VHDL syntax rule for constraints is also modified to include buffer constraints:

constraint ::=
    range_constraint
  | index_constraint
  | buffer_constraint

A buffer constraint may only be used in a subtype indication denoting a subtype of an unconstrained bounded channel type or in a constrained bounded channel definition. The simple expression in a buffer constraint must be a non-negative integer; it determines the buffer size for channels of the channel type or subtype. We also allow use of the 'length attribute to determine the buffer size for a constrained bounded channel type or an object of a constrained bounded channel type.

Example

```
-- unbounded channel type
type acknowledgment_channel is null channel;

-- constrained bounded channel types
type blocking_request_channel is
    channel buffer 4 of request_message;
type bigger_request_channel is
    channel buffer 2 * blocking_request_channel'length
of request_message;

-- unconstrained bounded channel type
type result_channel is
    channel buffer <> of request_message;
-- constrained bounded channel type
subtype blocking_result_channel is
    result_channel buffer 2;
```

———

The rules covering declaration of channel objects are largely unchanged from our original proposal. However, a restriction on the subtype indication in a channel-object declaration is that it denote an unbounded channel type or a constrained bounded channel type. A declared channel object may not be of an unconstrained bounded channel type. (This corresponds to the existing VHDL rules governing use of unconstrained types.)

Example

```
channel acknowledgment : acknowledgment_channel;
channel request : blocking_request_channel;
channel result_1 : result_channel buffer 1;
channel result_2 : blocking_result_channel;
```

———

The static semantic rules covering declaration of interface channel objects are extended as follows. A formal interface channel can be declared of any of the three classes of channel types. Where the formal channel is of an unbounded channel type, the associated actual channel must also be of an unbounded channel type. This ensures that, if inferences are made about process behaviour based on the premise that sending to the formal channel does not block, those inferences remain valid independent of characteristics of the actual channel. Where a formal channel is of a constrained bounded channel type or subtype, the associated actual channel must be of the same constrained bounded channel type or subtype, and have the same buffer size as the formal channel. Where a formal channel is of an unconstrained bounded channel type, the actual channel must be of a constrained subtype of the formal's type, and the buffer size of the formal channel is inferred from the buffer size of the actual channel.

Example

In the following model fragment, pipe_link is an unconstrained bounded channel type. The process pipe_stage

has a generic constant size that is used to specify the buffer sizes for the formal input and output channel ports. Use of the generic constant in this way ensures that the channels have the same buffer size. The channel objects link1 and link2 are declared to be of an anonymous subtype of pipe_link, with buffer size link_buffer_size. In the process instance stage1, the generic constant is given the value link_buffer_size, and so the two channel ports assume that value for their buffer sizes. Hence the association with the actual channel objects is legal, since they have the same buffer sizes. This scheme is analogous to the way in which generics and signal ports of unconstrained array types are often used in standard VHDL.

```
type pipe_link is channel buffer <> of link_data;

process pipe_stage is
    generic ( size : natural );
    port ( channel link_in : in pipe_link buffer size;
           channel link_out : out pipe_link buffer size );
end process pipe_stage;

channel link1, link2 : pipe_link buffer link_buffer_size;
...
stage1 : process pipe_stage
    generic map ( size => link_buffer_size )
    port map ( link_in => link1, link_out => link2 );
```

3.1 Communication Statements and Blocking

The introduction of bounded buffering in communication channels now implies that a sender might block upon execution of a send statement. This occurs if the message buffer of the target channel is full. More precisely, if the target channel has buffer size *n*, a sender blocks if there is a receiver on the channel for which

$$\text{number of sent messages} - \text{number of received messages} = n$$

The sender remains blocked until all receivers have received the first of the previous *n* sent messages. In the case where the message buffer is unbounded, the sender never blocks. However, an implementation may run out of buffer capacity and thus not be able to continue execution of the model. In the case where the message buffer size is zero, synchronous communication results, with blocking semantics similar to those of CSP [8]. A sender blocks at a send statement unless all receivers are already blocked waiting for a message to arrive on the channel. Message transfer occurs directly between the sender and the receiver(s) only when all have arrived at their respective communication statements. Hence, the communication event forms a barrier at which the communicating parties synchronize.

In our previous language design, we only included receive alternatives in select statements, since they were the only communication statements at which a process could block. Since a process can block at a send statement in our revised language design, it is appropriate to extend the select statement to allow inclusion of send statements. The revised syntax rules are:

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

guard ::= when condition =>

select_alternative ::=
     receive_alternative
   | send_alternative

receive_alternative ::=
    receive_statement [ sequence_of_statements ]

send_alternative ::=
    send_statement [ sequence_of_statements ]
```

(Further revisions to the select statement are described in Section 4.) The dynamic semantics of the revised form of select statement are similar to the semantics described in Section 2.2. Determination of open alternatives is identical. Choice of an open alternative for execution now depends on whether the alternative is a receive alternative or a send alternative. A receive alternative can be executed if there is a message in the message buffer of the source channel, or, in the case of a channel with a buffer size of zero, if there is a sender waiting to send to the channel. A send alternative can be executed if the message buffer of the target channel is not full, or, in the case of a channel with a buffer size of zero, if all of the receivers of the channel are waiting to receive from the channel. If none of the open select alternatives can be executed immediately, the process blocks until one of the open select alternatives can be executed. (As described in Section 4, this differs slightly from the semantics of our original proposal.)

Example

The following example illustrates inclusion of blocking send and receive alternatives in a select statement. The process models a network interface that accepts packets from a source and forwards them in a stream over a network. Flow control is modeled by the finite buffer size of the channel type (packet_stream_channel) representing the network stream. The network is assumed to be unreliable, so the receiver (not shown), periodically sends acknowledgment messages that include the sequence number of the last correctly received packet. The network interface

process saves packets until they have been been acknowledged. If an acknowledgment message indicates incorrect reception of a packet, the network interface process resets the packet save buffer back to the last correctly received packet and retries sending from the incorrectly received packet.

```
type packet_source_channel is
    channel buffer 0 of packet_type;
type packet_stream_channel is
    channel buffer window_size of packet_type;
type ack_stream_channel is
    channel buffer 1 of ack_type;

process network_interface is
    port ( channel packet_source :
                    in packet_source_channel;
           channel packet_stream :
                    out packet_stream_channel;
           channel ack_stream :
                    in ack_stream_channel );
    . . .
begin
    select
        not full(save_buffer) =>
            receive incoming from packet_source;
            insert(incoming, save_buffer);
    or
        send next_outgoing(save_buffer)
            to packet_stream
        advance(save_buffer);
    or
        receive (ok, last_seq_no) from ack_stream
        if ok then
            reclaim(last_seq_no, save_buffer);
        else
            reset(last_seq_no, save_buffer);
        end if;
    end select;
end process network_interface;
```

———

4. Timeouts for Message Reception

We address the lack of a timeout mechanism in the communication features of SUAVE by extending the select statement to include a timeout alternative. The revised syntax rule, extended beyond that described in Section 3, is:

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

timeout_alternative ::=
    timeout_guard sequence_of_statements

timeout_guard ::= after time_expression =>
```

The timeout alternative, if present, specifies the maximum amount of time for which the process will remain blocked waiting for an open select alternative to be executed. If the process remains blocked for the specified time after commencing execution of the select statement, the process resumes and executes the sequence of statements in the timeout alternative, thus completing execution of the select statement. If the timeout alternative is omitted, the process may block indefinitely.

Example

The following statements show how a real-time process might request information from a server. If the server does not respond before the process's deadline, the process proceeds without the response.

```
send request_details to server_request;
select
    receive response_info from server_response;
    act_on(response_info);
or after 10 ms =>
    act_on(default_info);
end select;
```

———

Addition of the timeout alternative to the select statement caused us to reconsider the semantics of the **else** clause, and in particular, to consider the meaning of including both a timeout alternative and an **else** clause in a given select statement. In our previous proposal, the **else** clause is essentially overloaded with two semantic mechanisms. It is used to handle the case of no alternatives being open, and also to handle the case of no open select alternatives being immediately ready to execute. The previous proposal did not provide a means of distinguishing between these cases.

The introduction of the timeout clause in this proposal allows us to provide cleaner semantics for the **else** clause and to differentiate between the two cases just mentioned. We thus revise the semantics of the **else** clause as follows. As before, if no select alternative is open and the **else** clause is present, the statements in the **else** clause are executed, completing execution of the select statement. It is an error if no select alternative is open and there is no **else** clause. If there are open select alternatives but none of them is ready to be executed immediately, the process blocks. In that case, if there is a timeout clause, it determines the maximum amount of time for which the process will remain blocked. If the timeout clause has a timeout interval of 0 fs, the process will resume on the next simulation cycle.

Example

The following example illustrates the use of a timeout alternative in a select statement to choose an alternative action when an output channel is full. The model describes a lossy message source for a network system. The system (not shown here) has an input channel that can accept messages at a given maximum rate. The channel has a bounded buffer to absorb bursts of messages that arrive at a greater rate. However, if the buffer capacity is exceeded, the message source discards messages.

```
type bounded_channel is
    channel buffer max_size of message_type;
process message_source is
    port ( channel message_stream :
                        out bounded_channel );
    variable next_message : message_type;
begin
    . . . -- construct next message in stream
    select
        send next_message to message_stream;
        . . . -- log successful send
    or after 0 fs =>
        . . . -- log loss of message from stream
    end select;
end process message_source;
```

While the timeout mechanism we propose is similar to the use of SDL timers described in Section 2.3, there are important differences. SDL timers are very general, and can also be used for generating events that occur at particular times. However, their use for message timeouts requires careful insertion of timer-set statements in all state transitions that end in the state in which a timeout is required, and timer-reset statements in all non-timeout state transitions from the state. By contrast, the timeout mechanism proposed for SUAVE is significantly simpler to use, and hence less prone to erroneous use. Other timing applications, such as measurement of message interarrival times, can be implemented using standard VHDL features such as wait statements, calls to the function "now" and variables of type "time."

5. Conclusion

Design at the system level requires use of abstraction to manage complexity. SUAVE extends VHDL to provide a more abstract form of communication between processes and components in a model. Our revised language design for communication features provides greater scope for formal verification of high-level models. The addition of timeout features also makes the language applicable to a wider class of modeling problems, including modeling of telecommunication protocols. In revising our language design, we have continued to avoid bias towards hardware or software refinement of designs. This is a significant strength of the language, allowing it to be used to express behavior and structure of a system before partitioning into hardware and software. Thus the language continues to support exploration of hardware/software trade-offs and hardware/software co-design, though now for a wider range of applications.

Work is in progress to implement the language extensions within the SAVANT framework [19], and validation experiments are planned, involving use of the language by industrial partners for real-world designs.

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 and P. A. Wilsey, "Extensions to VHDL for Abstraction of Concurrency and Communication," *Proceedings of Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98)*, Montreal, Canada, pp. 301–308, 1998.

[5] 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.

[6] 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.

[7] 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.

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

[9] G. J. Holzmann, "The Model Checker Spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[10] G. J. Holzmann, "State Compression in Spin," *Proceedings of Third Spin Workshop*, 1997.

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

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

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

[14] A. Parashkevov and J. Yantchev, "ARC—A Verification Tool for Concurrent Systems," *Proceedings of Third Australasian Parallel and Real-Time Conference*, Brisbane, Australia, 1996.

[15] A. Parashkevov and J. Yantchev, "Space Efficient Reachability Analysis Through Use of Pseudo-Root States," *Proceedings of Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97*, Twente, The Netherlands, 1997.

[16] A. W. Roscoe, "Modelling and verifying key-exchange protocols using CSP and FDR," *Proceedings of IEEE Symposium on Foundations of Secure Systems*, 1995.

[17] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood, "Hierarchical compression for model-checking CSP or How to check 10^{20} dining philosophers for deadlock," *Proceedings of First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95*, Aarhus, Denmark, pp. 133–152, 1995.

[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.